

USENIX

UNIX APPLICATIONS DEVELOPMENT SYMPOSIUM



**UNIX APPLICATIONS
DEVELOPMENT**

Symposium Proceedings

**Toronto, Ontario, CANADA
April 25-28, 1994**

**SPRING
1994**

For additional copies of these proceedings write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is \$15 for members and \$20 for nonmembers.
Outside the U.S.A. and Canada, please add
\$9 per copy for postage (via air printed matter).

1994 © Copyright by The USENIX Association
All Rights Reserved.

ISBN 1-880446-61-8

This volume is published as a collective work.
Rights to individual papers remain with the author or the author's employer.

USENIX acknowledges all trademarks herein.

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.



USENIX Association

**Proceedings of the
1994 USENIX**

**UNIX Applications Development
Symposium**

**April 25-28, 1994
Toronto, Ontario, CANADA**

Table of Contents

USENIX UNIX Applications Development Symposium
April 25-28, 1994
Toronto, Ontario, Canada

Technical Sessions Program Wednesday, April 27, 1994

Introductory Remarks
Jim Duncan, Pennsylvania State University

Keynote Address
Quelling Riots, Earthquakes and Godzilla – Porting SimCity to X11
Robert Adams, Dux Software

DEVELOPMENT TOOLS

Wrapping DCE/OSF Client/Server Applications..... 1
Israel Gold and Uri Shani, IBM Israel Science and Technology

Dagger: A Tool to Generate Program Graphs 19
Yih-Farn (Robin) Chen, AT&T Bell Laboratories

Developing Applications with a UIMS 37
Daniel Klein, Lonewolf Systems

IMPROVING THE ENVIRONMENT

SPP – Low Tech, Practical, UNIX Software Portability..... 57
John Sellens, University of Waterloo

Creating a Configurable Compiler Driver for System V Release 4..... 67
John F. Dooley and Vince Guarna, Motorola Computer Group

Jam – Make(1) Redux 79
Christopher Seiwald, Ingres Corporation

Thursday, April 28, 1994

SOLVING UNIQUE PROBLEMS

Writing, Supporting, and Evaluating Tripwire: A Publically Available Security Tool 89
Gene H. Kim and Eugene H. Spafford, Purdue University

Design, Distribution, and Management of Object Oriented Software 109
Arindam Banerji, David Cohn, and Dinesh Kulkarni, University of Notre Dame

NEW APPROACHES FOR X

Better Widget Design: A Practioner's Approach 121
Ed Lycklama, KL Group

Implementing a Generalized Drag-and-Drop in X 133
Cui-Qing Yang and Shrinand Desai, University of North Texas

The Xt Intrinsics as a General Purpose Application Development Platform or A User Interface Toolkit
With Optional Users 147
Jordan M. Hayes, Heuristicrats Research; Charles A. Ocheret, Investment Management Services

EXPERIENCES

Bridging the Technology Generation Gap: Upgrading a Network Management Application to a
New Technology Base..... 159
Jay S. Lark, Teknekron Communications Systems

Porting and Maintaining with X and Motif: A Retrospective View 171
Paul Davey, User Interface Technologies Ltd.

Software Design for Installability 177
Steve Simmons, Inland Sea

Program Co-Chairs

Jim Duncan, Pennsylvania State University

Greg Woods, Plan IX, Inc.

Program Committee:

Frank Byrum, Digital Equipment Corporation

Neil Groundwater, SunSoft

Rob Kolstad, Berkeley Software Design, Inc.

Evan Leibovitch, Sound Software

Peter Renzland, Ontario Government

Greg Rose, RoSecure Software

Dan Tomlinson, Compusoft

Elizabeth Zwicky, SRI International, Inc.

PREFACE

For all the work that's gone into consistency and predictability, it's no doubt that Unix as testbed has led to manifold innovation at the expense of simplicity. Over the years, small armies of researchers have wrought new and interesting features into Unix and ported it into a wide variety of environments. The resulting complexity is both a boon and a bane, for while it offers untold flexibility, it also makes life very difficult for the poor souls who must develop, install, and maintain software.

To the typical software developer bringing a product from other desktop environments, Unix presents a daunting set of obstacles, ranging from platform independence to user interface consistency to multi-user processing and security issues to networked filesystems and transaction processing. Somewhere in this collective body of Unix software, however, are the kernels of knowledge about how to do things the "best" way. That's what this symposium is all about.

The program committee has chosen fourteen papers to be presented on a wide variety of topics all relevant to Unix applications development. The papers range from theoretical to down-to-earth, but in the USENIX tradition, they all include demonstrable, practical results. We're very lucky to have our keynote speaker, Robert Adams of Dux Software, here to tell us how and why the port of SimCity to Unix didn't work the first time, nor the second time, for that matter. And our invited speaker, David Tilbrook of Mortice Kern Systems, will tell us why it's easy to write a program if you already know how.

A lot of people have put a lot of work into producing this symposium. We'd like to mention a few, and regret that we can't include everyone. We'd like to thank Ellie Young, Judy DesHarnais, and Rob Kolstad for freely giving of their time and opinion; Carolyn Carr for doing a great job with the proceedings; Dan Klein, Richard Stevens, and Rob again for organizing and presenting the tutorials; Robert Adams and David Tilbrook for being our guest speakers; and of course, the authors of the papers, without whom we wouldn't have a symposium. The program committee read and evaluated a lot of papers in a very short time; we thank them for their time and acknowledge the support which they have received from their employers. Last but not least, the USENIX staff are wonderful people, and they are the single most important reason that the USENIX Association is a success.

Jim Duncan, Penn State University
Greg Woods, PlanIX, Inc.

AUTHOR INDEX

Arindam Banerji	109
Yih-Farn (Robin) Chen	19
David Cohn	109
Paul Davey	171
Shrinand Desai	133
John F. Dooley	67
Israel Gold	1
Vince Guarna	67
Jordan M. Hayes	147
Gene H. Kim	89
Daniel Klein	37
Dinesh Kulkarni	109
Jay S. Lark	159
Ed Lycklama	121
Charles A. Ocheret	147
Christopher Seiwald	79
John Sellens	57
Uri Shani	1
Steve Simmons	177
Eugene H. Spafford	89
Cui-Qing Yang	133

Wrapping DCE/OSF Client/Server Applications

Israel Gold and Uri Shani

*Haifa Research Laboratory
IBM Israel Science and Technology
Matam — Advanced Technology Center
Haifa 31905, Israel*

E-mail: igold@vnet.ibm.com, shani@vnet.ibm.com

Abstract. DCE is a comprehensive RPC-based solution for client/server applications across networks of heterogeneous machines. Unfortunately, DCE is hard to learn and use. The fact is, that even simple DCE applications may have a rather complicated structure, requiring a good understanding of the elaborate DCE technology. This paper presents *gluegen* — an automatic tool for “wrapping” plain C code to become part of a distributed application. The approach taken separates DCE-specific from DCE-independent elements of the application, via a high-level specification language. This language provides an easy and flexible way to describe the distributed application topology, and frees the programmer from having to get into the gory details of the DCE run-time.

Using *gluegen*, the development of simple DCE applications remains a rather simple task, requiring very little knowledge of DCE, and of the DCE run-time. The tool is particularly useful for *splitting* existing monolithic programs into clients and servers, with almost no change to the original application code. Complex DCE applications which do not neatly fit splitting are supported as well.

1 Introduction

Distributed Programming Environment (DCE) by the Open Software Foundation (OSF) offers a comprehensive solution for distributed client/server applications across networks of heterogeneous multi-vendor machines [8]. First to enjoy this support are the UNIX variants from HP, IBM and Digital¹. The DCE services and development toolkit come in several levels based at the bottom on *Threads* and *Remote Procedure Call* (RPC) [1]. *Threads* allow concurrent threads of control to execute within the same application process. DCE RPC is based on Apollo's NCS [5, 6], providing a synchronous communication mechanism which appears at the application level as a local procedure call.

Above these basic layers come additional services: *Directory Services* maintain a global symbolic identification system of servers on the network; *Security Services* provide authentication and

¹ UNIX, HP, IBM, and Digital are registered trademarks of the respective companies.

authorization services in an open network environment; *Time services* maintain global time synchronization, and a *Distributed File System*. Each component of DCE comes from a well established and proven technology, all integrated into a single coherent environment (see references in [8]).

The DCE run-time complexity can be appreciated by the size of its library — a few hundred functions to support all the above services. Close to a hundred support the fundamental RPC level. The full comprehension of DCE is a significant task, realizing that DCE applications may have a rather complicated structure, even for simple cases.

To develop a DCE application, you first have to design its distributed architecture, and write DCE code to perform RPC operations. The minimal requirement on the server side amounts to *registering* and *unregistering* its remote services. On the client side this amounts to locating and *binding* to servers and calling remote services. In order to perform RPC to remote services, they must be defined as operations in an *Interface Definition Language* (IDL) file. The *idl* compiler generates *stubs* from IDL files which are C sources that are compiled and linked with the application to facilitate and perform RPC for it. We assume in this paper that IDL files are written by the programmer.

We look at the situation where the application can have a monolithic version — where no elements of distributed programming exist. The program is then *split*, or *partitioned* into several partitions playing the role of clients, servers, or both. If properly done, the application can coexist in its monolithic version as well as its split version².

The main aspect of splitting existing applications as addressed in this paper is to keep DCE-related operations in a file separate from the original sources, thus keeping them as DCE-independent as possible. This is workable since DCE RPC calls can be performed transparent to the client (caller) and the server (triggered) code. DCE provides an *implicit handle* option which maintains the same function signature of a remote function as a local function. Even for the alternative *explicit handle*, which adds an extra parameter to a remote function, the separation of many aspects of DCE from the application source contributes appreciable modularity to the distributed program.

The *gluegen* tool introduced in this paper generates *glue-code* (“wrapping” code) which establishes the DCE working environment around plain C code, making it part of a distributed application. The glue-code takes care of properly binding the client and server by performing all the necessary interaction with the DCE run-time so that RPCs will succeed.

Glue-code is generated based on a small high-level specification language we term the *Application Profile* (APF) language, which frees the programmer from getting into the gory details of DCE run-time.

The rest of the paper is organized as follows: Section 2 presents a simplified view of the DCE application models that *gluegen* can handle. Section 3 describes informally the APF language aspects limited to RPC level support only. Section 4 provides a client/server component overview in *gluegen* environment. Section 5 discusses APF language extensions related to DCE Security and Directory services. And finally, Section 6 concludes with a short comparison to other

² DCE cannot support this approach to its fullest since there are aspects of program partitioning [9] which DCE cannot comply with. For instance, how to split a global variable. However, a significant simplification of DCE application development can be achieved with program splitting.

approaches in the field. Appendix A presents a complete example for DCE application development using *gluegen*.

2 DCE Application Models

The code generated by *gluegen* allows DCE applications to startup, establish communication (binding handle), and perform RPC. To clarify how *gluegen* works, we first present a simplified view of the possible DCE application models which *gluegen* can handle.

DCE building blocks for distributed applications allow the development of rather complicated and intricate solutions. An operational DCE application can be represented in the general case as a dynamic graph consisting of nodes and edges. Nodes are application programs executing on (possibly) different machines, and edges are DCE *interfaces* used by these applications.

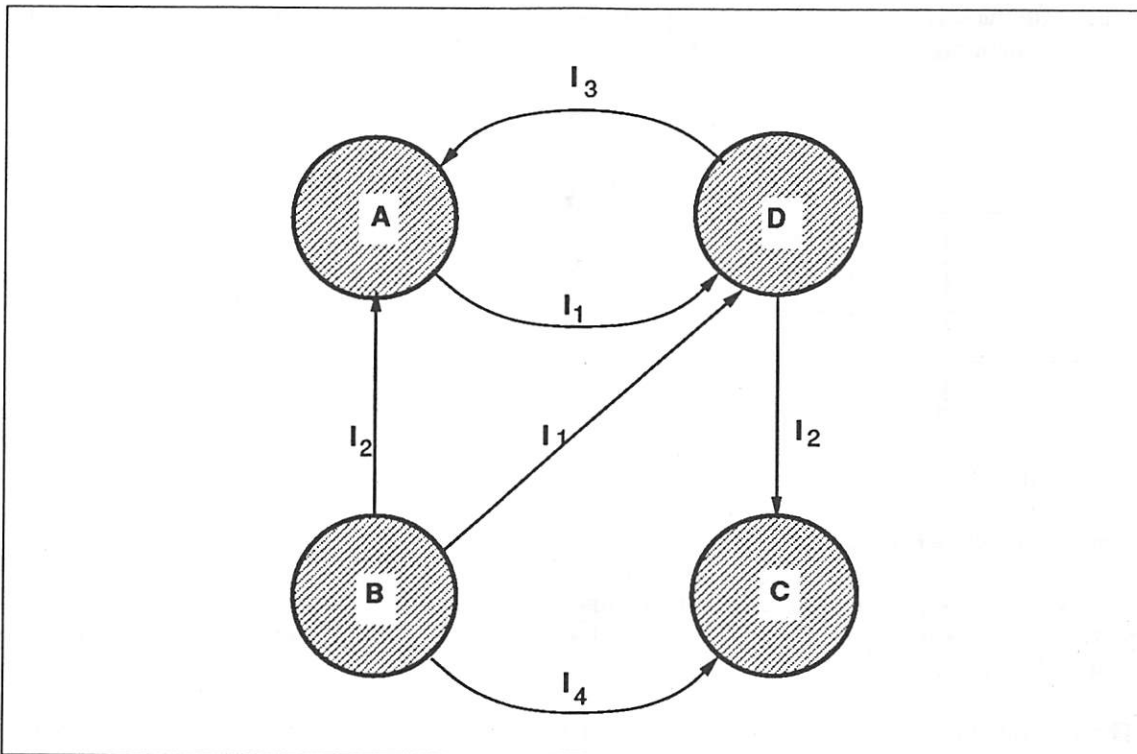


Figure 1. Distributed Application Graph. An edge in the graph represents a collection of RPC-able operations and is named after the DCE Interface describing these operations. The edge arrowhead indicates the direction of an RPC. An edge I_2 directed from node B to node A specifies that interface I_2 is *imported* by node B (source of edge) and *exported* by node A (target of edge). A single application node can import/export multiple interfaces.

A DCE *interface* is a collection of RPC-able operations, defined by a particular IDL file. When an application node imports a given interface, it is said to be a *client* of that interface; when it exports a given interface, it is said to be a *server* of that interface. An application node cannot import and export the same interface.

The graph edges are named after the DCE interfaces they represent, and their arrowheads indicate the direction of RPC calls; an edge I_2 directed from node B to node A specifies that interface I_2 is

imported by node *B* (source of edge) and exported by node *A* (target of edge). Figure 1 depicts a schematic representation of this idea.

The topology of an application can change during execution. Nodes may appear, establish communication, disappear, or move (i.e., be assigned to different hosts at different times). The most general situation can only be implemented by coding a program in a general-purpose language which uses the DCE/RPC primitives.

Although the general task of DCE programming is very broad, we are looking for a formal description of this task so that generic templates can be prepared ahead of time and be used to easily build useful DCE applications. In a nutshell, we have two major goals: First to define the details of a single interface (edge) between DCE nodes of which the IDL file is a major component, and secondly to combine multiple interfaces into DCE applications.

Given a distributed application graph, it can be described by an **Interface Mapping Table** that lists which interface is exported/imported by each application node. For example, Figure 2 is the interface mapping table of the distributed application graph in Figure 1.

Interface	Application Node			
	A	B	C	D
<i>I</i> ₁	Im	Im		Ex
<i>I</i> ₂	Ex	Im	Ex	Im
<i>I</i> ₃	Ex			Im
<i>I</i> ₄		Im	Ex	

Legend:

Im — Imported interface
Ex — Exported interface

Figure 2. Interface mapping table.

Observe that a single application node may import and/or export multiple interfaces. Moreover, a particular interface may be exported by several application nodes. In our example, interface *I*₂ is exported by nodes *A* and *C*.

The distributed application topology is defined by means of a small high level specification language we term *Application Profile* (APF) language. Application profiles written using the APF language are used by *gluegen* to configure DCE applications.

3 Application Profile Language

In this section we give an informal description of the APF language, by showing examples of simple client/server application profiles using the language. For clarity we only discuss here the language aspects that are related to RPC level support; Directory and Security Services are introduced in Section 5.

The APF language identifies a set of attributes which completely define how the DCE application should interact with the DCE run-time. These attributes are organized in an *interface profile* and an *application profile*. An *interface profile* identifies a DCE interface; it is associated with a partic-

ular IDL file, and as a result, with a unique universal identifier (UUID) and a set of RPC operations. The interface profile defines the binding method for the given DCE interface. The interface name and versions of the corresponding IDL file are automatically extracted by *gluegen*.

An *application profile* combines a collection of interface profiles for either *import* or *export*. The application profile also defines how the application will be initialized (start up), and how many concurrent threads of service will coexist in a server.

APF attributes fall into two categories: *compile-time* attributes and *run-time* attributes. Values of compile-time attributes must be specified in the APF file. Values of run-time attributes may be specified, or modified, at run-time upon invocation of the client and server applications. Values of run-time attributes can come from the command-line, the standard-input, or another file — depending on the user's choice in the APF. Likewise, values of attributes which are needed by clients, can be reported by the server to the standard output streams, or to a file. This too is done according to the user's choice in the APF file. When the server reports its binding attribute values to a file, the file can then be fed to the client. As a result, the two applications will bind in a very simple and direct method which DCE terminology would categorize as *string-binding*.

3.1 Example

Consider the distributed application graph in Figure 1. Application node *A* imports interface *I₁* and exports interfaces *I₂* and *I₃*. Definition of application node *A* in APF file APPA.APF, might look as follows (keywords are in bold letters):

```
/* File APPA.APF - Application Profile for Node A */

interface I1 {
    protseq = ncadg_ip_udp;
    host = node0;
    bindtype = repm;
    handle = explicit;
    idl = "I1.idl";
}

interface I2 {
    protseq = ncadg_ip_udp;
    bindtype = lepm;
    handle = explicit;
    idl = "I2.idl";
}

interface I3 {
    protseq = ncadg_ip_udp;
    bindtype = lepm;
    handle = explicit;
    idl = "I3.idl";
}

application appA {
    finput = null;
    foutput = stdout;
    nthreads = 1;
    import I1;
    export I2;
    export I3;
}
```

Explanation:

The application profile defines an application named *appA*. The binding attributes for *appA*, some provided with command arguments and some resolved at run-time, will be written to the standard output (*foutput* = stdout). No binding attributes will be read from input file (*finput* = null). The application will work serially using a single thread (*nthreads* = 1).

All three interfaces use the same protocol sequence (*protseq* = ncadg_ip_udp). The exported interfaces *I₂* and *I₃* will be registered by the local entry-point mapper — the DCE *rpcd* daemon. The imported interface *I₁* will bind *appA* to a server on remote host *nodeD* (*bindtype* = repm, and *host* = nodeD). All three interfaces use an explicit handle (*handle* = explicit). Each interface also has its own IDL file.

To make the application *appA*, invoke *gluegen* as follows:

```
gluegen APPA.APF appA
```

This will generate a pair of object files: glue-code file named *appA.o*, and glue-stub file named *appA_gstub.o*. To get the C source modules, rather than the compiled object files, use the *-keep c_source* option on the command line.

3.2 Application Profile for Simple Client and Server

Consider the case of client and server applications running on the same host, where the application *Server* exports interface *I₁* and the application *Client* imports it. The AFP definitions for the two applications might look as follows:

```
/* Application Profile for simple Client and Server
** running on the same host
*/
interface I1 {
    protseq = ncadg_ip_udp;
    bindtype = lepm;
    handle = implicit;
    idl = "I1.idl";
}

application Server { export I1 }

application Client { import I1 }
```

4 Glue-Code Generation

It is convenient to view the *gluegen* tool as parallel to the *idl* compiler (see Figure 3) when “wrapping” an application to operate properly in the DCE environment.

The *gluegen* generated code has three components:

glue-code — Generated by *gluegen* from an APF *application profile*, and consists of the application *main()* entry point. This is a simple program which makes very few calls to routines in the *glue run-time library*, described below.

glue-stub — Generated by *gluegen* from an APF *application profile*, building internal structures where all the APF information is stored, and where references are defined for global variables declared in the DCE stubs generated by the *idl* compiler. This portion can be viewed as parallel to the RPC stubs generated by the *idl* compiler³.

glue run-time library — Supports the glue-code and glue-stub generated by *gluegen*. It has a similar role as the DCE run-time library which supports the RPC stubs generated by the *idl* compiler. The glue-library makes use of the DCE run-time library.

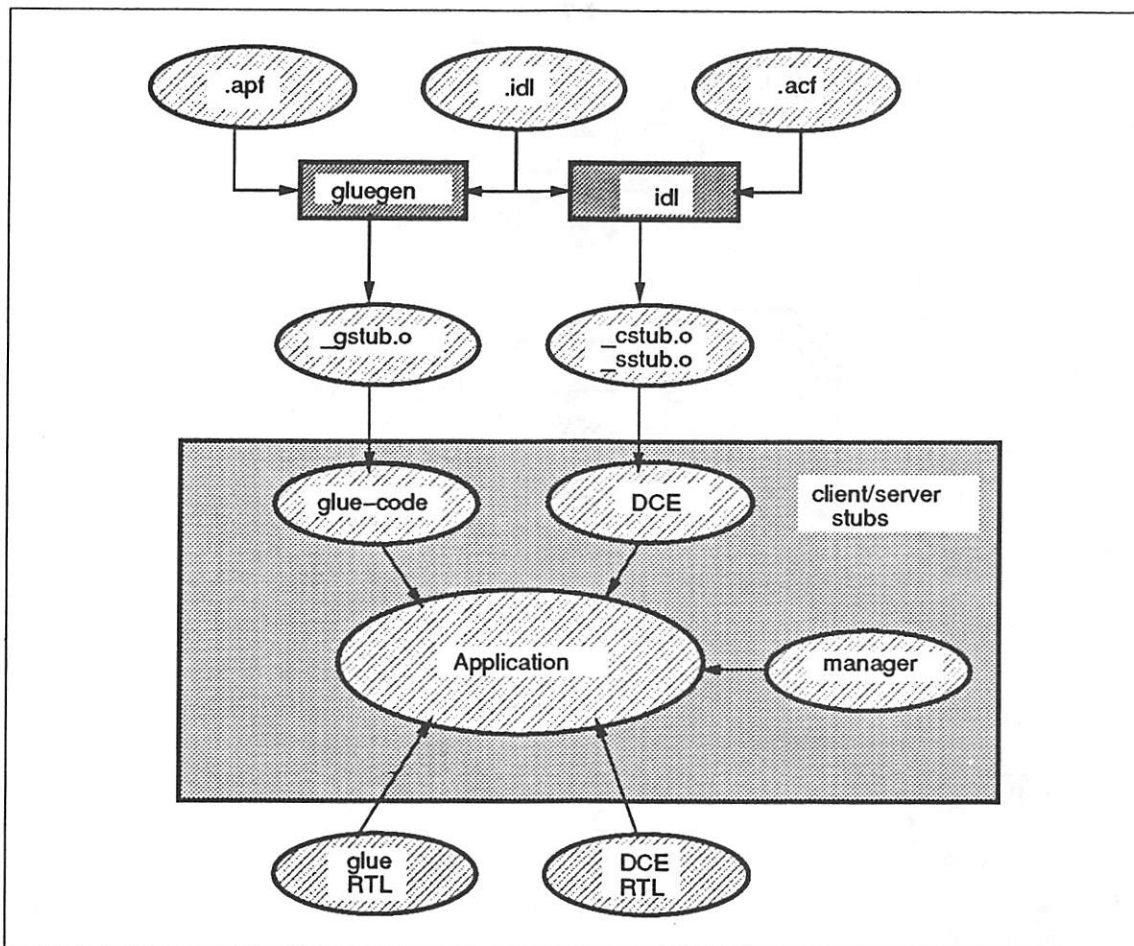


Figure 3. GLUEGEN and IDL compilers relationship.

4.1 Putting It All Together

To demonstrate the relationships and origin of components which make up a DCE application using *gluegen*, see Figure 4 for a situation where a monolithic application (top) is *split* to two programs serving in the roles of a client and a server (bottom).

³ Observe that *gluegen* generates a single glue-stub for each application, whereas *idl* generates multiple DCE stubs - one for each interface in the application. Also note, that as side effect of glue-stub generation, *gluegen* creates an ACF file for each imported interface that uses an implicit handle.

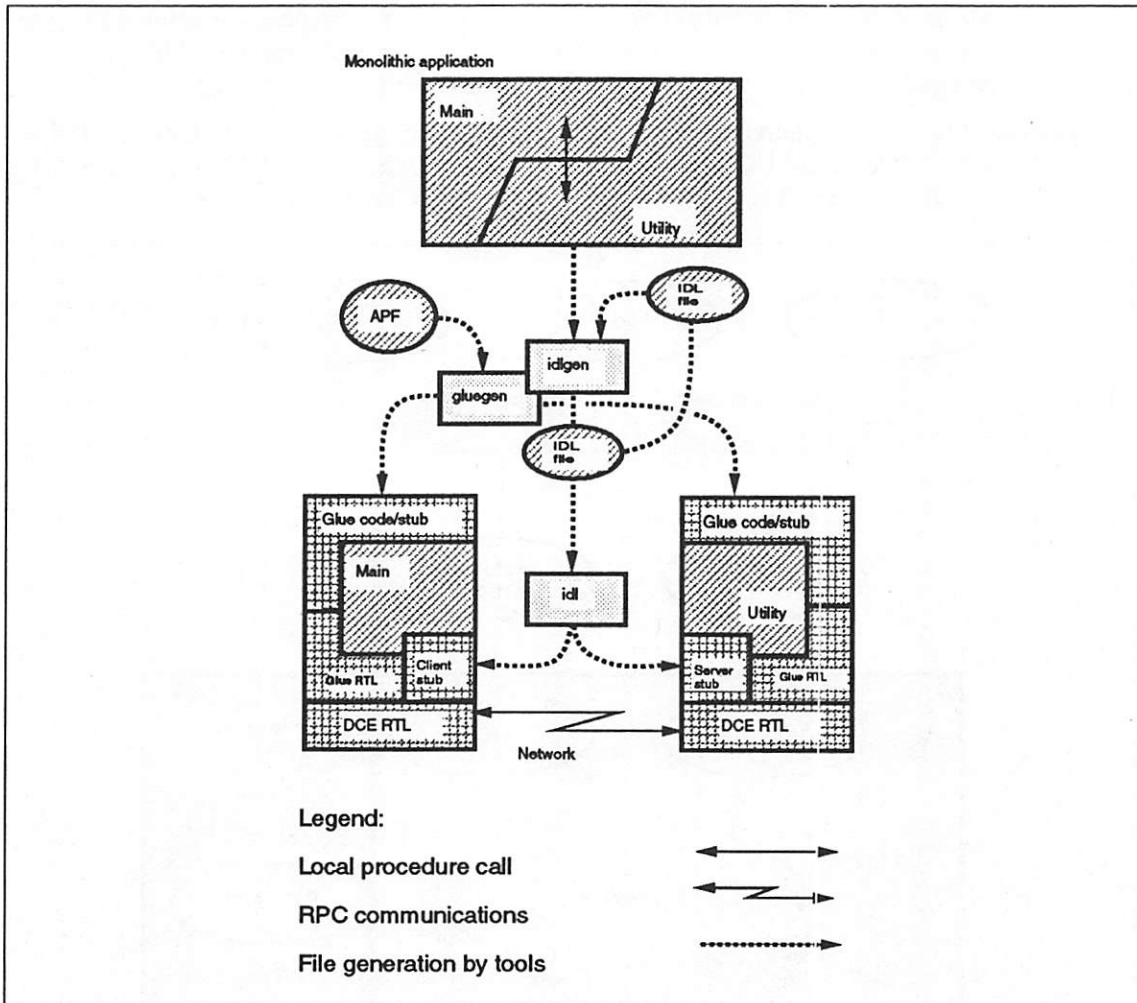


Figure 4. Putting it all together

The components of each application node (Client and Server) are as follows:

1. Glue-code and glue-stub are generated by *gluegen* from an APF file. Separate pairs of glue-code and glue-stub files are generated for each application — one for the client, and one for the server. The glue-codes include the *main()* entry-points of the client and server programs.
2. The glue-library is linked with each program and implements the API used in the glue-code. Application code may also use the API of the glue-lib.
3. Application code in the client is the *Main* part of the original application — which performs an RPC to the Server.
4. Application code in the server is the *Utility* part of the original application — which implements the RPC performed by the client.
5. Server-stub is a code generated by the *idl* compiler (part of DCE) for the server. The code is generated from an IDL file which represents the interface between the client and the server.
6. Client-stub is a code generated by the *idl* compiler for the client.
7. The IDL file used to generate the DCE stubs is extracted from the C source by another tool, *idlgen* (not described here), which can be considered for all purposes as if done manually — the common way of writing IDL files for DCE.

8. DCE RTL is the DCE run-time library which supports the DCE execution environment of DCE applications. This layer uses other lower-level communication support layers in the operating systems of the respective platforms on which the application nodes execute.

Appendix A describes the application development steps using *gluegen*. The appendix includes step by step instructions for converting a simple application (the *binop* program) into a DCE Client/Server application.

5 DCE Security and Directory Services

In this section we introduce the extension of the APF language to support DCE Security and Directory Services. Three new profiles with their specific attributes are introduced: *login profile*, *security profile*, and *object profile*.

The *login profile* is used to setup an application login context. The *security profile* is used to associate authentication and authorization service with an imported interface. The *object profile* is used to specify a directory service object that will be used by servers to export their binding information, and by clients to import that information.

5.1 Setting a Login Context

Individual DCE applications run under a *login context* which contains the credentials of a particular DCE *principal*. Any program that wishes to use DCE directory and security services must establish a DCE login context prior to using them.

Normally, users establish a DCE login context using the `dee_login` program. Executing `dee_login` is necessary if an application uses the DCE directory service, regardless of whether it intends to use the security service as well.

The login context credentials do not last forever, each principal may have a different lifetime for its credentials. After this time is up the principal credentials expire and the login context must be refreshed. In such circumstances, it may be also advisable to refresh the principal security key (password), if it has been updated while the application is running.

Setting up and maintaining a DCE login context for application *A* is controlled by the *login profile* that is associated with the *application profile* for *A*. For example, the following login profile will use a given security key-table (`keytab = /home/server/keytab`) to setup a login context for the "binop" server (`principal = ./servers/binop`).

```
/* Login Profile for creating and refreshing a login context
** for the "binop" server, using a given key table
*/
login keytab_login {
    principal = "./servers/binop";
    keytab = "/home/servers/keytab";
    refresh_login = yes;
    refresh_sec_key = yes;
}
```

The `refresh_login` and `refresh_sec_key` attributes direct *gluegen* to generate glue-code that will fire two DCE threads: one for refreshing the login context and one for refreshing the security key. If no *login profile* is associated with the application, the *login context* is inherited from the environment.

5.2 Authentication and Authorization

DCE security includes *authentication* and *authorization* services which provide for secure RPC. The authentication service enables both client and server, possibly on different machines, to validate each other's (principal) identity. The authentication service is done automatically by the DCE run-time on behalf of both client and server, using whatever authentication level they have defined.

Servers must also determine whether an authenticated client can be granted *authorization* for the RPC it requests. Servers normally do this by checking the authentication level used by the client and verifying the client's privilege to access a particular directory service entry.

The APF language allows clients to specify a separate authentication service for each interface; that is to associate different authentication services with different sets of callable RPC operations. For example, the *security profile* below guaranties that each operation in *binop.idl* will be serviced by the binop server (*c_server_princ_name* = *././servers/binop*). The specified authentication level directs the DCE run-time on the server side to verify that none of the client's RPC arguments was modified or forged (*c_protect_level* = *pkt_integ*).

```
/* Binding Security Profile for binop clients */

security binop_security {
    c_authn_svc = rpc_c_authn_dce_secret;
    c_authz_svc = rpc_c_authz_dce;
    c_server_princ_name = "././servers/binop";
    c_protect_level = pkt_integ;
}

interface binop {
    protseq = ncadg_ip_udp;
    bindtype = lepm;
    handle = implicit;
    idl = "binop.idl";
    security = binop_security
}
```

The authentication level for RPC calls (*c_protect_level* attribute) may assume the following values:

- default** — Use the default protection level for the specified authentication service
- none** — Perform no data protection
- connect** — Perform data protection only when the client establishes communication with the server (first RPC call)
- call** — Perform data protection on each RPC
- pkt** — Ensure that all data received is from the expected client
- pkt_integ** — Ensure and verify that none of the data transferred between the client and the server has been modified.
- pkt_privacy** — Perform protection on every packet and also encrypt each RPC argument value

Server authentication parameters are associated with its application profile. This is due to the fact that server authentication can only be registered on a per-principal basis. A server can specify three security parameters: its requested authentication service, *s_authn_svc*, its requested

authentication level, **s_protect_level**, and its home directory service entry for checking client access privileges, **nspath**. This is demonstrated by the following application profile.

```
/* Application Profile with Authentication and Authorization */
```

```
security binop_security {
    c_authn_svc = rpc_c_authn_dce_secret;
    c_authz_svc = rpc_c_authz_dce;
    c_server_princ_name = "/./:/servers/binop";
    c_protect_level = pkt_integ;
}

interface binop {
    protseq = ncadg_ip_udp;
    bindtype = lepm;
    handle = implicit;
    idl = "binop.idl";
}

login keytab_login {
    principal = "/./:/servers/binop";
    keytab = "/home/servers/keytab";
    refresh_login = yes;
}

application MyServer {
    s_authn_svc = rpc_c_authn_dce_secret;
    s_protect_level = pkt_integ;
    nsopath = "/./:/servers/binop";
    login = keytab_login;
    export binop;
}

application MyClient {
    import binop { security = binop_security }
}
```

For secure communication, servers must implement an authorization policy. To assist the programmer we have implemented the **IFIsClientAuthorized()** function which returns **FALSE** if the client does not use the proper level of authentication, or has no access permission to the server's home directory service entry (defined by **nspath**). This function is contained in the glue runtime library and is published in **glue.h** — the glue-code header file.

5.3 Directory Service Objects

DCE Directory Service add another whole world of possibilities, specifically in hiding elaborate details of *objects* (resources) within symbolic names in Global Directory Service (GDS) and Cell Directory Service (CDS) data bases. DCE objects are uniquely identified by a UUID and are strongly related to binding clients and servers. Directory entries associate an object with a name and other attributes. A typical attribute for RPC servers is their host (machine) location.

An object can be considered as an *implementation* of an interface, and there may be several different implementations of the same interface which are distinguished by an object UUID. Moreover, multiple identical implementations of the same interface may coexist on several machines on the network.

In order for a client to properly bind to its servers, it needs to perform certain activities for each imported interface. First, it has to identify all the objects of the imported interface. Second, it imports all the server binding handles stored in these objects, and selects the candidate servers on

the network according to some criteria. Servers, on the other hand, need to export their binding handles to all objects associated with their interfaces.

The APF language associates an object with a given interface using the *object profile*. For example, the following APF defines a server for the "Line Printer" interface *I_lp*.

```
/* Interface Profiles with its associated Object */

login keytab_login {
    principal = "/./servers/printers";
    keytab = "/home/servers/keytab";
}

object objLP {
    objid = uuid(0007b93a-7a67-1cf3-90fa-10005aa8b716);
    nse = "/./server/printers/LP";
    ns_export = yes;
    ns_unexport = yes;
}

interface I_lp {
    handle = explicit;
    idl = "lp_print_if.idl";
    object = objLP;
}

application lp_server {
    login = keytab_login;
    export I_lp
}
```

The object identified by *objLP* is a directory service entry named *"/./servers/printers/LP"* (*nse* = *"/./servers/printers/LP"*). The object is associated with interface *I_lp* which is exported by the *lp_server* application. The *lp_server* will export its binding handle to *objLP* on each invocation (*ns_export* = yes), and will remove it from *objLP* on completion (*ns_unexport* = yes).

6 Conclusions

This paper introduces the *gluegen* tool for wrapping C code into DCE applications, with respect to splitting monolithic applications into clients and servers. The tool serves two goals: reduce code dependency on DCE, and relieve the programmer from having to be deeply familiar with the details of DCE development toolkit.

There are many alternative approaches to distributed application development. DCE is not an object-oriented system, even though its internal architecture is. DCE is intended for developing procedural applications on a distributed environment. Object oriented alternatives [7] may or may not use DCE as an implementation base. For instance, there are efforts to enrich DCE with objects [4], or to build a C++ library to encapsulate the DCE run-time API [3]. A similar approach is distributed free of charge by the Citibank Distributed Processing Technology [2]. Both approaches show examples of how DCE applications become simple and small with their high-level objects. With *gluegen*, the same applications becomes **much** simpler. Moreover, applications also have a comparable *stand-alone* monolithic version to go along. In addition, *gluegen* uses a library which introduces a higher-level abstraction above the DCE run-time, while keeping it at the procedural format. The run-time support in *gluegen* strongly relates to the elements of the APF language, where *applications*, *interfaces*, and *DCE objects* are treated — at the language

and in the internal representation — as objects in the “OOP” sense. *gluegen* run-time library maintains this model and gives access to it through a limited API (not included in this paper).

Our approach keeps DCE aspects separated from the application logic and makes it much less dependent not only on DCE, but also on the fact that the application is distributed.

A totally different approach is to introduce a new language [10] where aspects of distribution are language elements integrated with the application logic. The application is then independent of DCE, which now is only one choice of an implementation vehicle in the language. Nevertheless, wrapping aspects have also been introduced into that system, through a separate language from the application source. We believe that the approach adopted by DCE of providing support for distribution via functions is preferred. Independence of the application from DCE can be achieved by separating application logic from aspects of distribution as two orthogonal implementation efforts. Our tool offers an essential instrumentation in this direction.

7 Acknowledgements

Thanks to Arie Tal, and Karen Laster for help in developing and running the regression tests.

8 References

- [1] Birrell, A. and B., Nelson, “Implementing Remote Procedure Calls,” *ACM Trans. on Computer Systems*, vol. 2, pp. 39-59, Feb. 1984.
- [2] Citibank Distributed Processing Technology, Objtran Programmer’s Guide (Available via internet from lcp@fig.citib.com), 1993.
- [3] Dilley, J., “Object_Oriented Distributed Computing With C++ and OSF DCE,” *Proceedings of the International DCE Workshop, A. Schill (ed.)*, pp. 256-266, Karlsruhe, WG: Springer-Verlag, October 1993.
- [4] Mock, M. U., “DCE++: Distributing C++ Objects using OSF DCE,” *Proceedings of the International DCE Workshop, A. Schill (ed.)*, pp. 242-255, Karlsruhe, WG: Springer-Verlag, October 1993.
- [5] Network Computing Architecture, Apollo Computer Inc., Prentice Hall, 1991.
- [6] Network Computing System Reference Manual, Apollo Computer Inc., Prentice Hall, 1991.
- [7] Object Management Group, The Common Object Request Broker: Architecture and Specification, 1991.
- [8] Open Software Foundation, DCE Application Development Guide, 1993.
- [9] Shani, U., N., Amit, I., Boldo, M., Kaplan, J., Marberg, R. Y., Pinter and M., Rodeh, “Program Partitioning for Heterogeneous Machines,” *Proceedings, The Sixth Israeli Conference on Computer Systems and Software Engineering*, pp. 136-145, Herzliyah, Israel, June 2-3 1992.

- [10] Yemini, S., G., Goldszmidt, A., Stoyenko, Y., Wei and L., Beeck, "Concert: A Heterogeneous High-Level-Language Approach to Heterogeneous Distributed Systems," *Proceedings of the 9th International Conference on Distributed Computing Systems*, 1989.

9 Appendix A - Development Steps of the Binop Application

The following section describes the application development steps using *gluegen*. Step by step instructions are shown for converting a simple application (the **binop** program) into a DCE Client/Server application.

program files:

main.c — main module of the binop program which calls the `binop_add()` function.

```
/*    binop program functions    */

void binop_add(a, b, c)
long a, b, *c;
{
    *c = a + b;
}
```

binop.c — module containing the `binop_add()` function.

```
/*    binop program main() entry point    */

#include <stdio.h>

main(int argc, char *argv[], char *envp[])
{
    char *msg = "Binop Application Completed";
    int i, n, pass, failures=0, PASSES=10, CALLS=10;

    for (pass = 1; pass <= PASSES; pass++) {
        printf("PASS (%d):", pass);
        for (i = 1; i <= CALLS; i++) {
            binop_add(i, i, &n);
            if (n != i+i) {
                printf("Two times %ld is NOT %ld\n", i, n);
                failures++;
            }
            printf(".");
        }
        printf("\n");
    }

    printf("%s: %d calls, %d failures\n", msg, PASSES*CALLS, failures);
}
```

Objectives: Create a DCE application in which the Client side will be based on `main.c` and the Server side will be based on `binop.c`.

9.1 Step 1: Generate IDL file for the Server side.

The first step is to generate the IDL file, **binop.idl**, describing the Server functions defined in **binop.c**. This may be done manually, or by using the **idlgen** tool developed by the authors as shown, below. Note that the DCE tool, **uuidgen**, creates a new UUID on each invocation.

```
User:    uuidgen -i | idlgen -stdin binop.c -ibd -interface binop > binop.idl
System:  ...
User:    cat binop.idl
System:
```

```
[
  uuid(0096BF68-065E-1BE6-B951-10005AA8B716),
  version(1.0)
] interface binop {

  ...
  /*@[export] binop_add;      file binop.c */

  void
  binop_add (
    [ in] long a,
    [ in] long b,
    [ out, ref] long *c
  );
}
```

9.2 Step 2: Create an APF file (.apf) for the Client and Server sides

The second step is to write an application profile, **binop.apf**, describing to *gluegen* the Client and Server sides of the distributed application. The contents of **binop.apf** might look as follows:

binop.apf file:

```
/* binop.apf : application profile defining
**    client and server applications for the "binop" program
*/
interface I1 {
  protseq = ncadg_ip_udp;
  bindtype = lepm;
  handle = implicit;
  idl= "binop.idl";
}

application server { export I1 }
application client { import I1 }
```

9.3 Step 3: Generate Glue-code and Glue-stubs using *gluegen*

Invoke *gluegen* to generate glue-code (DCE application programs) and glue-stubs for the *client* and *server* applications defined in **binop.apf**. To generate only the C source modules type the following.

```
gluegen binop.apf client -v -keep c_source
gluegen binop.apf server -v -keep c_source
```

The first *gluegen* command generates the client's glue-code and glue-stub, that is, **client.c** and **client_gstub.c**, respectively. Since the *client* application imports an interface that uses an implicit handle, *gluegen* will also generate an ACF file **binop.acf** (for **binop.idl**). The second *gluegen* command generates the server's glue-code and glue-stub, that is, **server.c** and **server_gstub.c**, respectively.

9.4 Step 4: Generate DCE stubs using the IDL compiler

Invoke the IDL compiler to generate DCE stubs for the Client and Server sides of the distributed application, using the IDL file **binop.idl** generated in Step 1. To generate only the C source modules type the following.

```
idl binop.idl -v -keep c_source -server none
idl binop.idl -v -keep c_source -client none
```

The first line generates the client's DCE stub **binop_cstub.c**; the second line generates the server's DCE stub **binop_sstub.c**.

9.5 Step 5: Convert your main program into a function.

The module **client.c** contains the new **main()** entry point for the Client side of the distributed application. Unless specified otherwise (using the **-fmain** option of *gluegen*), the old **main()** entry point in **main.c** is treated now as a function named **fmain()**. So, edit the C module **main.c** and convert the string "main" into "fmain." Write the edited file into a new file named **fmain.c**.

9.6 Step 6: Compile and link the Client Side

This step compiles and links the Client side of the distributed application. The Client source files are:

```
client.c      - main program generated by gluegen (Step 3)
client_gstub.c - glue-stub generated by gluegen (Step 3)
binop_cstub.c - DCE stub generated by IDL (Step 4)
fmain.c       - fmain() generated manually (Step 5)
```

In AIX type the following:

```
cc_r client.c fmain.c client_gstub.c binop_cstub.c -ldce -lglue -o client
```

9.7 Step 7: Compile and link the Server Side

This step compiles and links the Server side of the distributed application. The Server source files are:

server.c - main program generated by gluegen (Step 3)
binop.c - module containing the binop_add() function
server_gstub.c - glue-stub generated by gluegen (Step 3)
binop_sstub.c - DCE stub generated by IDL (Step 4)

In AIX type the following:

```
cc_r server.c binop.c server_gstub.c binop_sstub.c -ldce -lglue -o server
```

9.8 Step 8: Execute the distributed application

To run the distributed application invoke the **server** process in one window and the **client** process in another window. Since the binding method used in binop.apf is "lepm," the RPC daemon process, **rpcd**, must be running prior to invoking the server. Correct client output will look as follows:

```
User:     client
System:

      -protseq ncadg_ip_udp -host host-name -ep ep-number

PASS(1):.....
PASS(2):.....
PASS(3):.....
PASS(4):.....
PASS(5):.....
PASS(6):.....
PASS(7):.....
PASS(8):.....
PASS(9):.....
PASS(10):.....
Binop Application Completed: 100 calls, 0 failures
```

In the above, *host-name* stands for your host name and *ep-number* stands for the entry point number allocated by **rpcd** to the server. Running the stand alone binop program will give the same results, using the following commands to compile, link and execute the program.

```
cc main.c binop.c -o binop
binop
```


Dagger: A Tool to Generate Program Graphs

Yih-Farn Chen

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
chen@research.att.com

Abstract

Dagger is a tool that generates program graphs to selectively visualize a software structure. The design of dagger achieves a strong degree of language independence by exploiting the duality between a class of entity-relationship databases and attributed directed graphs. This paper describes the C and C++ versions of dagger, which map a selected subset of relationships stored in a program database to a graph specification, decorate the graph with display attributes, and then pass it to layout tools or interactive graph browsers. Dagger takes output from database queries to generate a large variety of interesting program graphs, including header file hierarchy, module binding, and type inheritance graphs. This paper describes the graph generation process surrounding dagger and a sample of tools involved in the process. In particular, we describe how a closure operator works in tandem with dagger to control graph complexity by generating reachability graphs where sub-structures are selectively ignored or expanded. Our experience in applying dagger to several software projects has demonstrated its capability in abstracting and visualizing complex software structures without much overhead. All program graphs presented in this paper are tagged with timing statistics.

1 Introduction

Programmers are frequently faced with the task of maintaining complex software systems with inadequate documentation. The situation is analogous to maintaining a complex building with inaccurate or missing blueprints. Tools that abstract and visualize such software systems to regenerate the necessary *software blueprints* can help programmers maintain the structure more effectively.

This paper presents a program visualization approach that focuses on automatically generating abstractions of the static code structure. The visualization process consists of a sequence of mappings. First, an *abstractor* analyzes the source of a program and maps it to a program database according to an entity-relationship model[3]. Next, a *query* or *operator* retrieves a subset of the database, which is then mapped to a directed graph. Finally, the graph specification is passed to layout tools or interactive graph browsers. The complete graph generation process is diagramed in Figure 1.

This paper describes **dagger**, a program visualization tool that controls the mappings from a program database to a directed graph specification. The design of dagger is largely language-independent because of its focus on entities and relationships. The program database used by dagger for C is generated by CIA[4][5], and the one for C++ is generated by CIA++[11][10]. The directed graph specification generated by dagger is acceptable by the graph-drawing tool **dot**[9][8]. By changing a set of mapping functions, dagger can easily adapt to different entity-relationship databases and automatic layout tools.

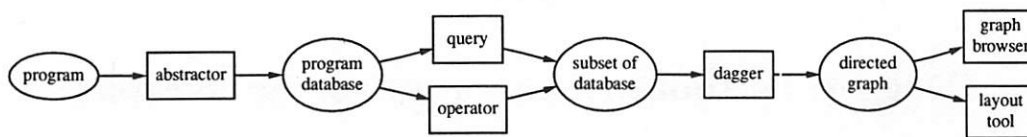


Figure 1: The process of program graph generation is a sequence of selective mappings

Dagger takes the output from database queries and operators designed for the CIA and CIA++ program databases to generate a large variety of program graphs, including header file hierarchy, entity dependency, module binding, type inheritance, reachability, and *focus* graphs (see Section 4). For example, Figure 2 shows the type dependency graph of a C program generated by dagger and drawn by dot with the following simple command pipeline¹:

```
$ dagger type - type - | dot -Tps
```

The graph generation process, including query execution, mapping to directed graphs, and generation of the PostScript[®] layout, took 3.88 seconds (1.20 seconds in user time + 2.68 seconds in system time) on a Series5/800 Solbourne[®] (a Sun4 clone) server running OS/MP 4.1A.3. All pictures in this paper are labeled with timing data collected on this machine. To generate this graph, dagger selects only type-to-type relationships. The picture reveals the mutual dependencies between the two types `struct _sfdc_` and `Sfdisc.t`. It also shows the relative structural complexity of `Ciarec.t` and `Symbol.t`.

Some abstractions such as reachable sets and program layers cannot be readily expressed by traditional database queries and must rely on special operators. For example, the CIA++ tool **Subsys** uses the closure construct in the Daytona data management system[12] to compute all entities and relationships reachable from a particular entity. Dagger is designed as an open-ended tool to optionally accept output generated by these operators to visualize their corresponding abstractions.

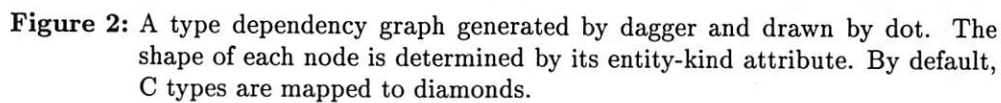
Roman and Cox define program visualization as a process of mapping programs to graphical representations and give a comprehensive survey on its various forms[18]. While static program visualization tools have become more available in recent program development environments such as ObjectCenter[®][15], SMARTsystem[®][17], and Energize[®][1], unlike dagger, they usually lack the abstraction power and flexibility needed to selectively visualize facets of complex structures in large software projects.

The rest of this paper is organized as follows: Section 2 gives a brief overview of the entity-relationship model of a C program database. Section 3 describes the mapping process of dagger followed by several examples of program graphs. Section 4 discusses how dagger can be used with a closure operator to provide interesting graphs beyond what typical queries can express. Section 5 describes our experience in applying dagger to large software projects and presents future directions in our program visualization efforts. Section 6 concludes with a summary.

2 An Entity-Relationship Database for C

Dagger is essentially a filter that takes as input a set of binary, directed relationships. To generate program graphs for C and C++ , it relies on information stored in the CIA and CIA++ program databases, which are based on the entity-relationship model. Both CIA and CIA++ record five kinds of non-local program entities (files, functions, types,

¹The `-Tps` option of `dot` sets the output language to PostScript[®].



```

/* file ftable.c */
#include "coor.h" /* defines types COOR and struct coor */
#define TBSIZE 2
extern COOR *rotate();
extern COOR *shift();
typedef COOR *(*PFPC)();
static PFPC ftable[TBSIZE] = { rotate, shift };

```

21

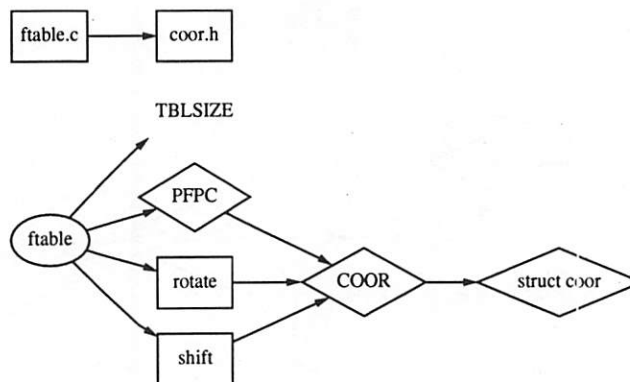


Figure 3: The complete program graph of a small C program. Files and functions are shown in boxes, types in diamonds, variables in ellipses, and macros in plain text.

files: `fable.c` and `coor.h`
 functions: `rotate` and `shift`
 types: `PFPC`, `COOR`, and `struct coor`
 variable: `fable`
 macro: `TBSIZE`

CIA also records nine relationships between these entities. For example, `fable` refers to the macro `TBSIZE`, the type `PFPC`, and two functions, `rotate` and `shift`, which in turn refer to the type `COOR`. In addition, the attributes (storage class, data type, definition/declaration, etc.) of each entity declaration are recorded in the database. In summary, the job of CIA is to selectively map program source text to an entity-relationship database, complete at its abstraction level, by ignoring certain details such as data and control flow information involving only local variables.

The complete database for the above piece of code can be easily mapped by dagger to a graph shown in Figure 3. However, for any real and complex programs, we must rely on abstractions to retrieve only a subset of all the relationships in a program database. In CIA and CIA++, abstractions can be obtained through queries or specialized program database operators. Queries for CIA and CIA++ are based on either `cql`[7] or `Daytona`[12]. They provide flexible access to information stored in the program databases. For example, the following `cref` query of CIA returns all functions referred to by the variable `fable`²:

```

$ cref var ftable func -
k1 file1      name1      k2 file2      name2
== =====
v ftable.c    ftable      p ftable.c    rotate
v ftable.c    ftable      p ftable.c    shift
  
```

All CIA tools (including dagger) that deal with relationships share this interface, where the arguments specify patterns for the two end points of a reference relationship: (*parent kind*, *parent name*) and (*child kind*, *child name*). All relationships that match the specified patterns are retrieved.

In order for other UNIX[®] commands (such as `awk`) and CIA tools (such as dagger) to further process the reference information, `cref` also generates unformatted output with

²In the CIA database, `p` is used as a shorthand for functions (procedures).

complete attributes for both the parent and child entities³:

```
$ cref -u var ftable func -
12;ftable;variable;0;ftable.c;PFPC [];static;7;0;7;def;3118aafa;; \
    6;rotate;function;0;ftable.c;COOR *;extern;4;0;4;dec;bed65a3a;;7
12;ftable;variable;0;ftable.c;PFPC [];static;7;0;7;def;3118aafa;; \
    10;shift;function;0;ftable.c;COOR *;extern;5;0;5;dec;e875f5f4;;7
```

This unformatted form of the cref output provides a standard data exchange format between many CIA tools, including subsys and dagger.

3 Generating Program Graphs with Dagger

Conceptually, dagger performs four steps during the mapping from a program database to a directed graph specification:

- *Selection*: Select a subset of relationships from a database by invoking queries or taking output from database operators.
- *Decoration*: Use the attribute values of entities or relationships to determine their corresponding display attributes, such as color, style, and shape.
- *Projection*: Select a subset of the entity and relationship attributes to filter out information unnecessary (such as the checksum of each entity) for applications of the final graphs.
- *Mapping*: Generate an attributed directed graph specification by mapping each entity to a node and each relationship to an edge. Graph attributes such as page size and aspect ratio are then added to control the graph layout.

The mapping functions in the decoration step are customized for each language and chosen to enhance the understanding of a program structure. For example, dagger maps different kinds of C or C++ entities to different shapes: file and function entities are mapped to box nodes, types to diamonds, variables to ellipses, and macros to plain text. In the case of C++ inheritance diagrams, the three possible values of the *access-specifier* attribute of each class inheritance relationship (public, protected, and private) are mapped to different edge styles: solid, dashed, and dotted. In addition, an edge is labeled “v” if the inheritance relationship is virtual. The decoration, projection, and mapping steps are straightforward, so we will focus on the selection step in this paper. The rest of this section and the next section give examples of graphs that are created through queries and operators, respectively.

While both CIA and dagger have been used to generate databases and graphs for projects that involve millions of lines of code, most program graphs presented in this paper are derived from the CIA program databases of two popular tools: **incl**[20] and **xgremlin**, an X11[®] version of **gremlin**[16]. Both programs have many users and have evolved over the years to meet user demands. Some metrics on the complexity of these two programs are shown in Table 1. The numbers include all the user and system header files used by these two programs, but they exclude the source files of libraries used. Both source and database sizes are measured in the number of bytes.

program	lines	source size	db size	entities	relationships
incl	1,957	49,963	26,416	392	517
xgremlin	24,582	620,726	390,430	4,842	6,634

Table 1: Complexity metrics for the two examples of program databases

³Due to typesetting constraints, each line of the actual output is split into two lines, separated by a backslash character.

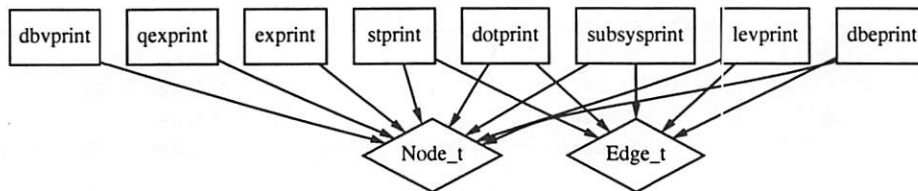


Figure 4: All printing functions in incl depend on Node_t, Edge_t, or both (generation time: 0.76u + 2.48s = 3.24 seconds)

Besides simple query patterns, dagger and cref support **ksh**[2]-like regular expressions, logical operators, and negations. For example, the following command generated a program graph of incl shown in Figure 4⁴

```
$ dagger -R func '*print' type - | dot -Tps
```

The diagram shows a selected subset of function-to-type relationships. Only functions whose names end with **print** and the types they refer to are displayed. The picture reveals an interesting structural fact that is not obvious in the corresponding relational view below: all functions that depend on Edge_t also depend on Node_t.

```
$ cref func '*print' type -
k1 file1          name1          k2 file2          name2
== =====
p incl.c          dbepprint      t incl.h          Node_t
p incl.c          levprint      t incl.h          Node_t
p incl.c          dbepprint      t incl.h          Edge_t
p incl.c          exprprint     t incl.h          Node_t
p incl.c          levprint      t incl.h          Edge_t
p incl.c          subsysprint   t incl.h          Node_t
p incl.c          qexprprint    t incl.h          Node_t
p incl.c          dotprint      t incl.h          Node_t
p incl.c          stprint       t incl.h          Node_t
p incl.c          subsysprint   t incl.h          Edge_t
p incl.c          dotprint      t incl.h          Edge_t
p incl.c          stprint       t incl.h          Edge_t
p incl.c          dbvprint     t incl.h          Node_t
```

Dagger can also be used to visualize array dependencies on macros, which are frequently used to set array sizes. The following command pipeline generates the graph shown in Figure 5, which shows all those array variables that depend on any one of the following macros **NFONTS**, **NSIZES**, or **N*STIPPLES**. The *selection clause* **file1=graphics.c** further limits our interests to only those arrays defined in **graphics.c**. The picture reveals that eight arrays satisfy the criteria and two of them depend on both **NFONTS** and **NSIZES**. These two arrays (**charysizes** and **font_info**) are two-dimensional.

```
$ dagger -R v - m 'NFONTS|NSIZES|N*STIPPLES' file1=graphics.c | dot -Tps
```

As an example for C++ program graphs, Figure 6 shows a typical type inheritance graph generated from a CIA++ program database. Note that the edge styles are adjusted according to the access specification of the inheritance relationship. **WorkTree** has a protected inheritance relationship with **Tree**, while **DispTree** has a private one. All other inheritance relationships are public and virtual ones are labeled with "v". The

⁴The -R option of dagger forces the picture to be drawn from top to bottom instead of left to right, the default direction.

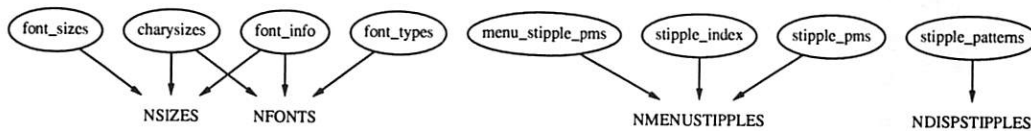


Figure 5: A graph that shows all variable arrays in `graphics.c` whose sizes depend on the selected macros (generation time: 1.28u + 2.23s = 3.51 seconds)

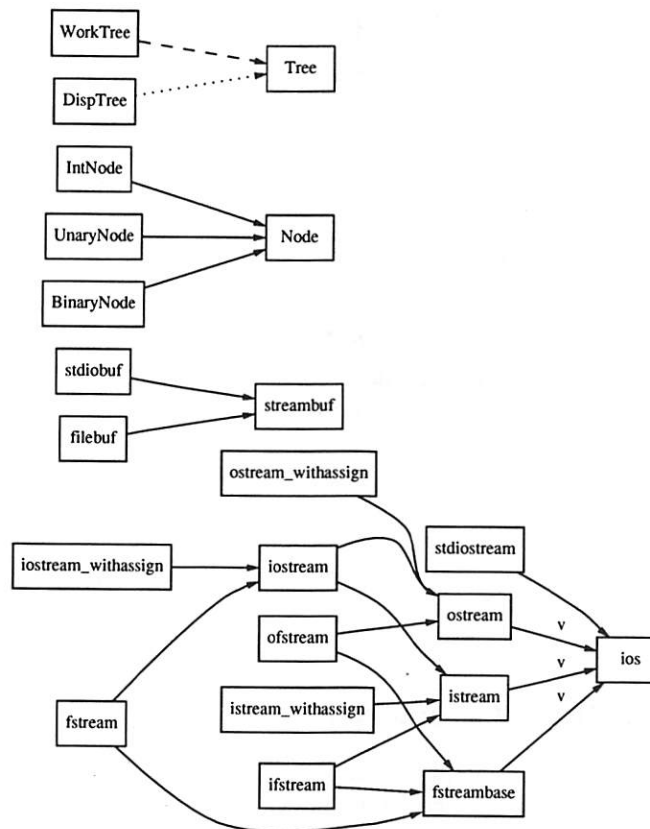


Figure 6: A C++ type inheritance graph (generation time: 1.98u + 4.66s = 6.64 seconds)

graph is generated by Dagger, the C++ version of dagger, with the following command pipeline, by limiting the relationship kind to inheritance (`rkind=i`):

```
$ Dagger type - type - rkind=i | dot -Tps
```

Note that the first two graphs in this section have only two layers because their parent and child entity kinds are different. The complexity of diagrams in this class of graphs can be managed easily by using regular expressions and selection clauses to narrow down the subset in each end. In the class of graphs where the parent and child entity kinds are the same, there tend to be many more layers because of their transitive nature. Examples include the type inheritance graph (type-to-type) shown in Figure 6, function reference graphs (function-to-function), and include hierarchies (file-to-file). Getting a good abstraction out of these graphs requires more than simple query manipulations.

As an example, Figure 7 shows the complex include hierarchies in `xgremlin` that are generated by the following command⁵:

```
$ dagger -F file - file '!(icons/*)' | dot -Tps
```

The diagram has been simplified by using a *negation* operator in the child name pattern to ignore all include relationships involving header files stored in the `icons` sub-directory. Unfortunately, the picture is still too difficult to read on a single page. What the user frequently wants is a single include hierarchy rooted at a particular source file with all other include relationships ignored.

While the labels in Figure 7 are hard to read, the function reference graph in Figure 8 generated by the following command seems to be beyond comprehension:

```
$ dagger -F func - func - | dot -Tps
```

The diagram does show some interesting facts, though:

- A few functions have very large fanins. Further studies on these functions can be performed with the CIA tool `ciafan`.
- Many functions do not seem to be on any of the reference paths from `main` (the root function). There are two possibilities: a) These are dead functions that never get exercised. The CIA tool `deadobj` does detect 11 dead functions in `xgremlin`. b) These are functions indirectly invoked through variables. In this case, both function-to-function and variable-to-function relationships (such as the one between `ftable` and `rotate` in the C example in Section 2), must be drawn to complete the picture. `Xgremlin` indeed has many functions defined in a variable array for menu selections. Section 5 discusses how dynamic function reference relationships can be captured for this class of programs during program execution.

A brute-force way to solve the graph complexity problem is to blow up the diagram to such an extent that all symbols become legible and use an option of `dot` to paginate the diagram. Magnifying complex graphs, however, is usually not satisfactory because the number of edge crossings increases substantially and they become a major distraction. In the next section, we show how `dagger` can work with a closure operator to limit graph complexity.

4 Managing Graph Complexity

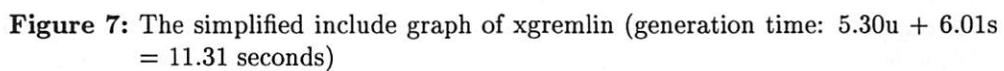
One way to manage the complexity of a large program graph is to concentrate on individual nodes of interest. Typical maintenance tasks might require

- Finding all program entities that an entity depends on directly or indirectly. This information is particularly useful for packaging software components and partitioning complex program graphs.
- Finding all program entities that might be affected by the change of an entity. `TestTube`[6], a system for selective regression testing, uses such information to determine what test cases must be rerun.
- Displaying a few layers of relationships centered around a particular node.

There is no easy way to specify such subsets of relationships using a simple query syntax. Fortunately, `dagger` is designed as an open-ended tool to optionally take output from any tools that follow the standard exchange format shown in Section 2. One such tool is the CIA closure operator `subsys`, which computes all entities and relationships reachable (either forward or backward) from selected entities. For example, the include hierarchy of `graphics.c` in `xgremlin` is generated by the following command pipeline⁶

⁵The `-F` option requests `dot` to stretch a picture and change its aspect ratio to maximize the use of all the space available within the specified area.

⁶The `-u` and `-e` options force `subsys` to generate the `cref -u` format.



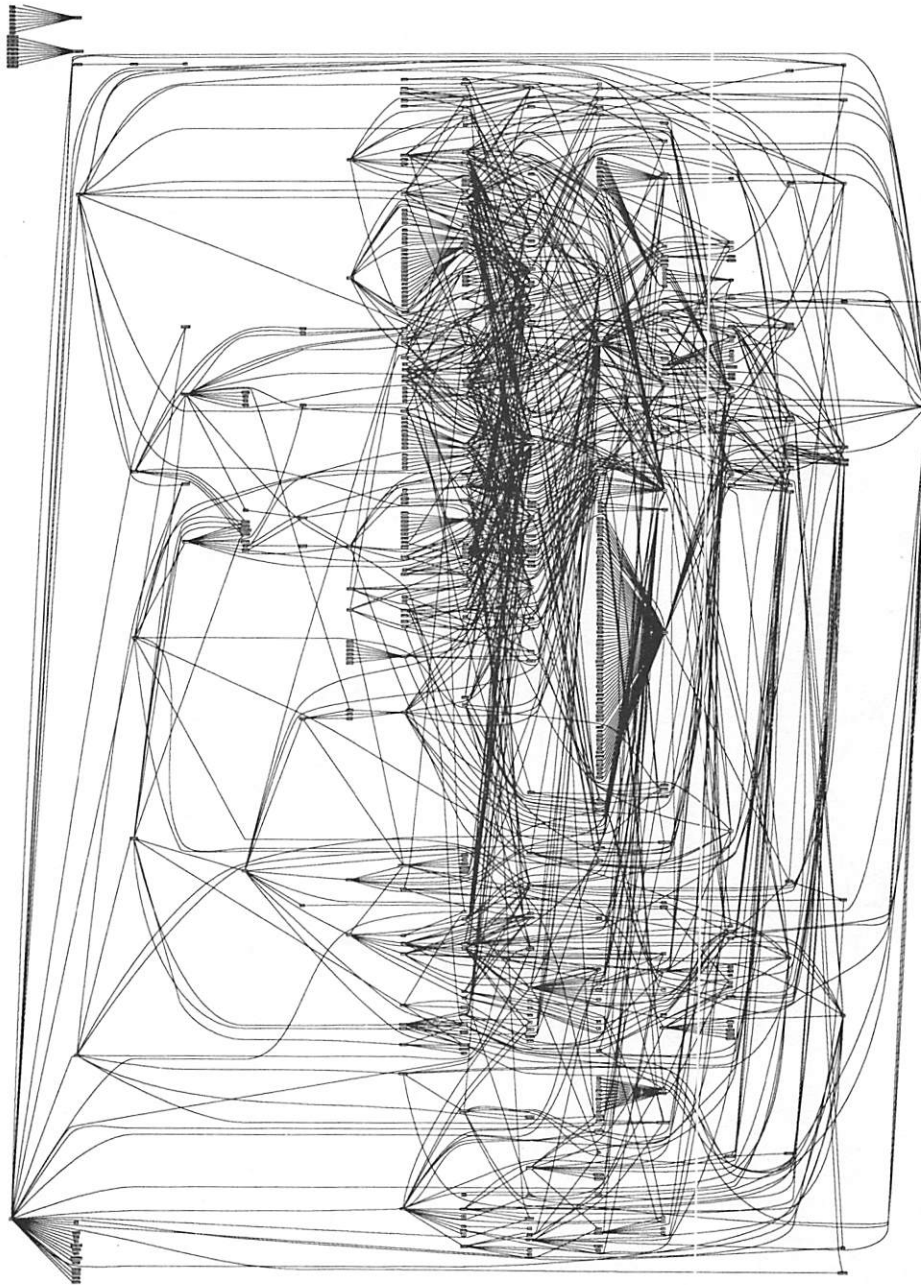


Figure 8: The complex function reference graph of xgremlin (generation time: 242.76u + 48.90s = 291.66 seconds)

and shown in Figure 9. The graph generation time includes the closure computation by `subsys`, the directed-graph generation by `dagger`, and the graph layout by `dot`.

```
$ subsys -u -e file graphics.c | dagger -i | dot -Tps
```

By default, `subsys` traces only reference relationships of *kind-to-kind*, where *kind* is the root entity kind. In the previous example, only file-to-file relationships are traced. However, it is frequently necessary to compute a *complete* reachable set, where all kinds of relationships are considered. For example, the following command (using the `-k` option)

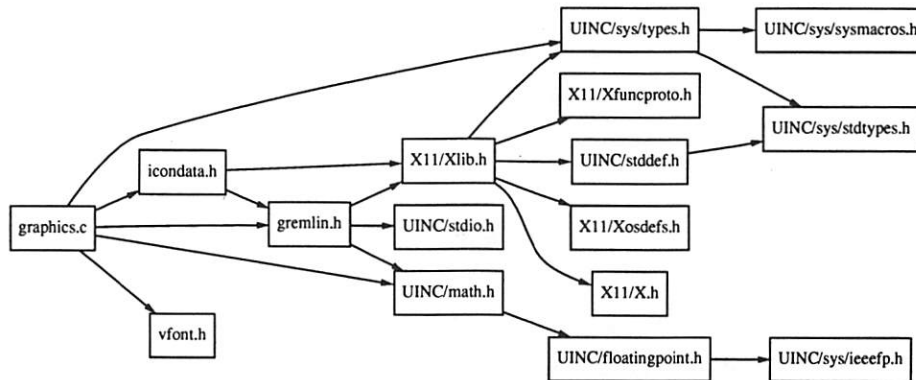


Figure 9: The include hierarchy rooted at `graphics.c` (generation time: 0.86u + 1.91s = 2.77 seconds)

generates a complete subsystem rooted at the function `mustuse` of `incl`. The results are shown in Figure 10.

```
$ subsys -u -k -e func mustuse | dagger -i | dot -Tps
```

Since `dagger` is just a filter, the set of graphs generated can be further extended by mixing an interesting collection of raw `cref` records and presenting the resulting set to `dagger`. The following simple shell script concatenates the output of two `subsys` commands and generates a *focus* graph with the specified entity at the center. The first `subsys` command traverses the graph starting from the root in a forward direction and the other one traverses backwards.

```
# Usage: focus [options] entity_kind entity_name
subsys -u -e "$@" > $$$.1
subsys -u -e -r "$@" > $$$.2
cat $$$.1 $$$.2
rm -f $$$.1 $$$.2
```

Figure 11 shows a focus graph centered at the `incl` function `mksymbol`, which is created by the following command. The `-l` option of `subsys` (and `focus`) limits the graph traversals to only two levels deep in each direction.

```
$ focus -k -l2 fu mksymbol | dagger -i | dot -Tps
```

Even the reachable set of a selected entity may be so complex that further simplifications are necessary. One common technique is to recursively partition a reachable set into trackable pieces. During the generation of a subsystem, we can stop the graph traversal at certain selected nodes, ignore their sub-hierarchies, and later expand these nodes as necessary. For example, Figure 12 shows the subsystem of the `xgremlin` function `GRArc` by ignoring the relationship hierarchy rooted at `GRVector` (with the `-i` option). The graph is generated by the following command pipeline, in which a simple `sed` script `markn` is used to change the color and style of the ignored node.

```
$ subsys -u -e -k -i GRVector fu GRArc | dagger -i | markn GRVector
| dot -Tps
```

It turns out that the subsystem of `GRVector` is still so complex that we have to ignore two entities during the graph traversal of itself: the two types `GC` and `Display`. Following is the command that generates the partial subsystem graph of `GRVector` shown in Figure 13. It is conceivable that another shell script can be written to generalize the partition process.

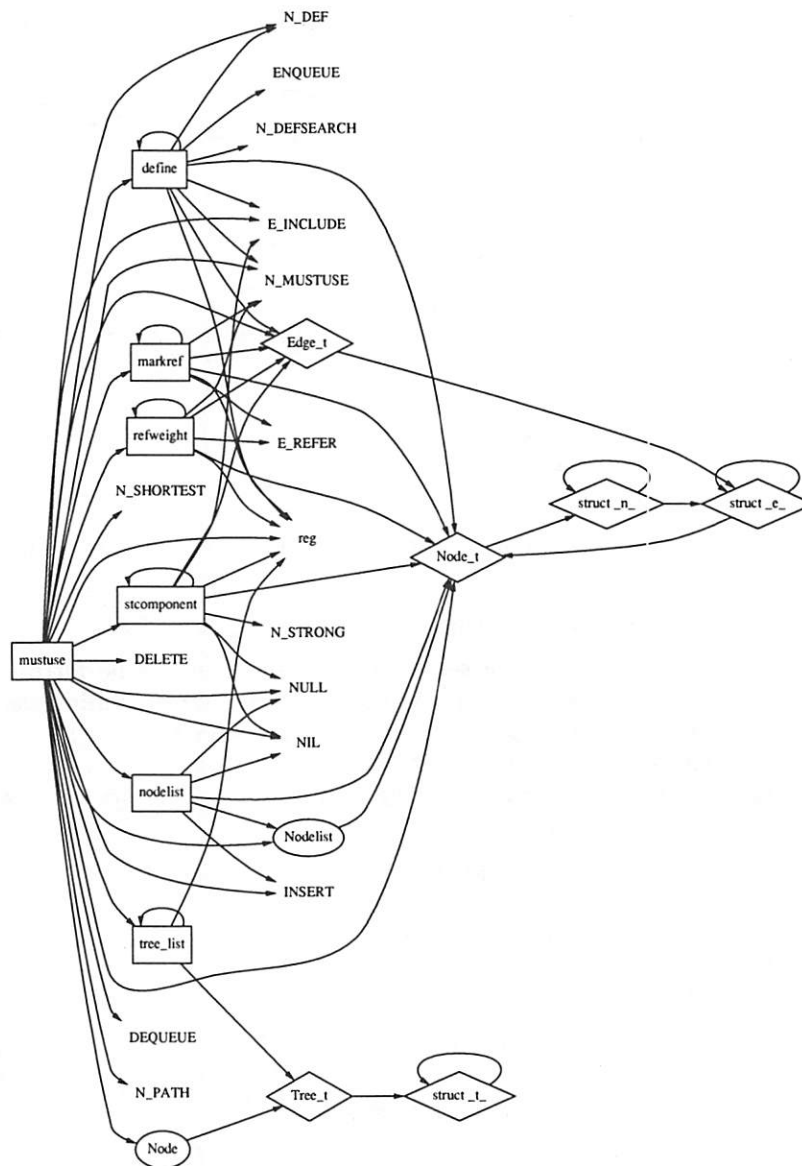


Figure 10: A complete subsystem graph rooted at the incl function `mustuse` (generation time: $2.66u + 2.36s = 5.02$ seconds). Functions are shown in boxes, types in diamonds, variables in ellipses, and macros in plain text.

```
$ subsys -u -e -k -i Display -i GC fu GRVector | dagger -i |
markn Display FC | dot -Tps
```

5 Experience and Future Directions

Dagger has been exercised by dozens of AT&T software projects. Many features in dagger and subsys, such as the style of mapping in the inheritance graphs and the partition process of subsys, are direct results of user feedback. The selective nature of dagger makes the generation of a variety of program graphs a simple task during software

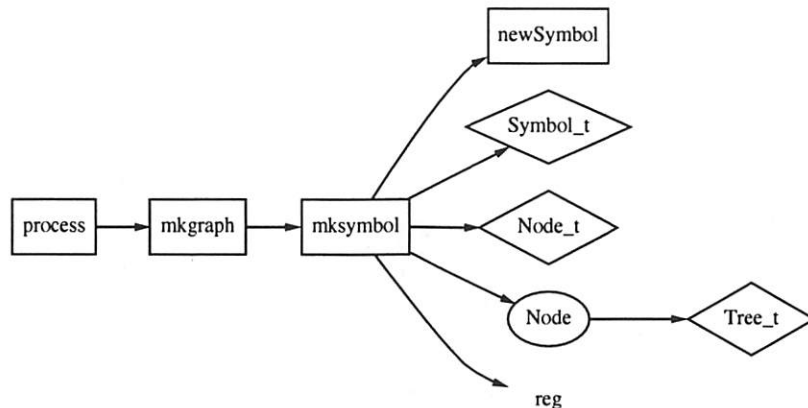


Figure 11: A five-layer focus graph centered at the incl function `mk-symbol` (generation time: $1.61u + 3.85s = 5.46$ seconds)

maintenance.

Several new directions in our program visualization efforts related to dagger and dot are emerging:

- *Cluster Graphs:*

The type dependency graph shown in Figure 2 can be modified by using an explicit, labeled cluster to group each set of types belonging to the same header file. Such a graph provides information on both header file and data structure dependencies. The notion of *subgraph* in dot is ideal for this purpose. A subgraph in dot is a subset of the graph's nodes and edges with its own graph attributes (such as its own graph label and default node and edge values). In general, dagger can use the value set of any entity attribute to create clusters. For example, one clustering approach is to group variables in a program graph according to their storage classes.

- *Records for Containment Relationships:*

Another desirable clustering method is to group all members of a C structure or a C++ class. A natural representation for the containment relationships between an aggregate structure and its members (which may themselves be aggregate structures) is dot's notion of *record*, a recursive array of labeled boxes. Dagger should be extended to visualize interactions between C++ class members using record nodes.

- *Undirected Graphs:*

So far, this paper discusses only directed program graphs, but some program structure information is best represented as undirected graphs. The CIA tool **share**, which is based on Daytona's query language Cymbal, computes the number of shared child entity references between any pair of parent entities. A high degree of sharing frequently implies that the two entities operate or depend on similar entities. Such a pair can be considered *strongly coupled* because a change in one entity is likely to affect the other. We can use the undirected-graph layout tool **neato**[14] to adjust the distance between any pair of nodes by mapping the coupling strength to the weight (or spring constant) of an edge. By tuning the mapping function, we attempt to create natural clusters where strongly coupled entities are drawn to each other.

- *Customized Mapping:*

Dagger uses pre-defined functions to map entity and relationship attributes in C and C++ to display attributes. This approach is not satisfactory because a program-

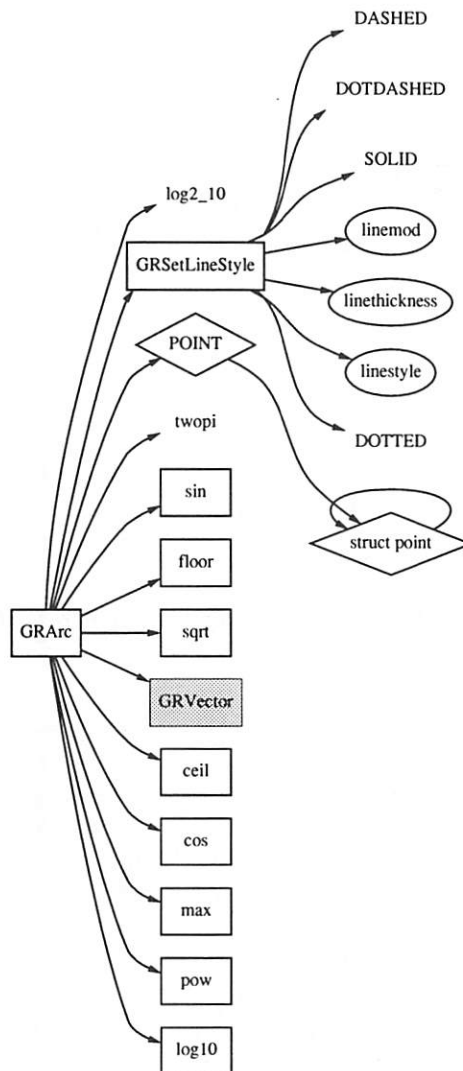


Figure 12: A partial subsystem graph rooted at the xgremlin function GRArc (generation time: $1.33u + 2.65s = 3.98$ seconds)

mer may decide to change the mapping to emphasize certain aspects of a program graph. For example, in a general C++ type dependence graph involving inheritance, friendship, and reference relationships, the edge styles (solid, dashed, dotted) may be used to represent these three types of relationships rather than the three kinds of access specifications (public, protected, and private) of inheritance relationships. We are considering a simple mapping mechanism based on dynamically changing cql's schema to support customized mapping.

- *Dynamic Program Graphs:*

The static function reference graph shown in Figure 8 only reveals the static function dependency relationships, which do not necessarily reflect the actual function calls during an instance of program execution. Work is underway in a program animation project to merge dynamic function reference information produced by a version of **app**[19], an annotation preprocessor enhanced with instrumentation capabilities,

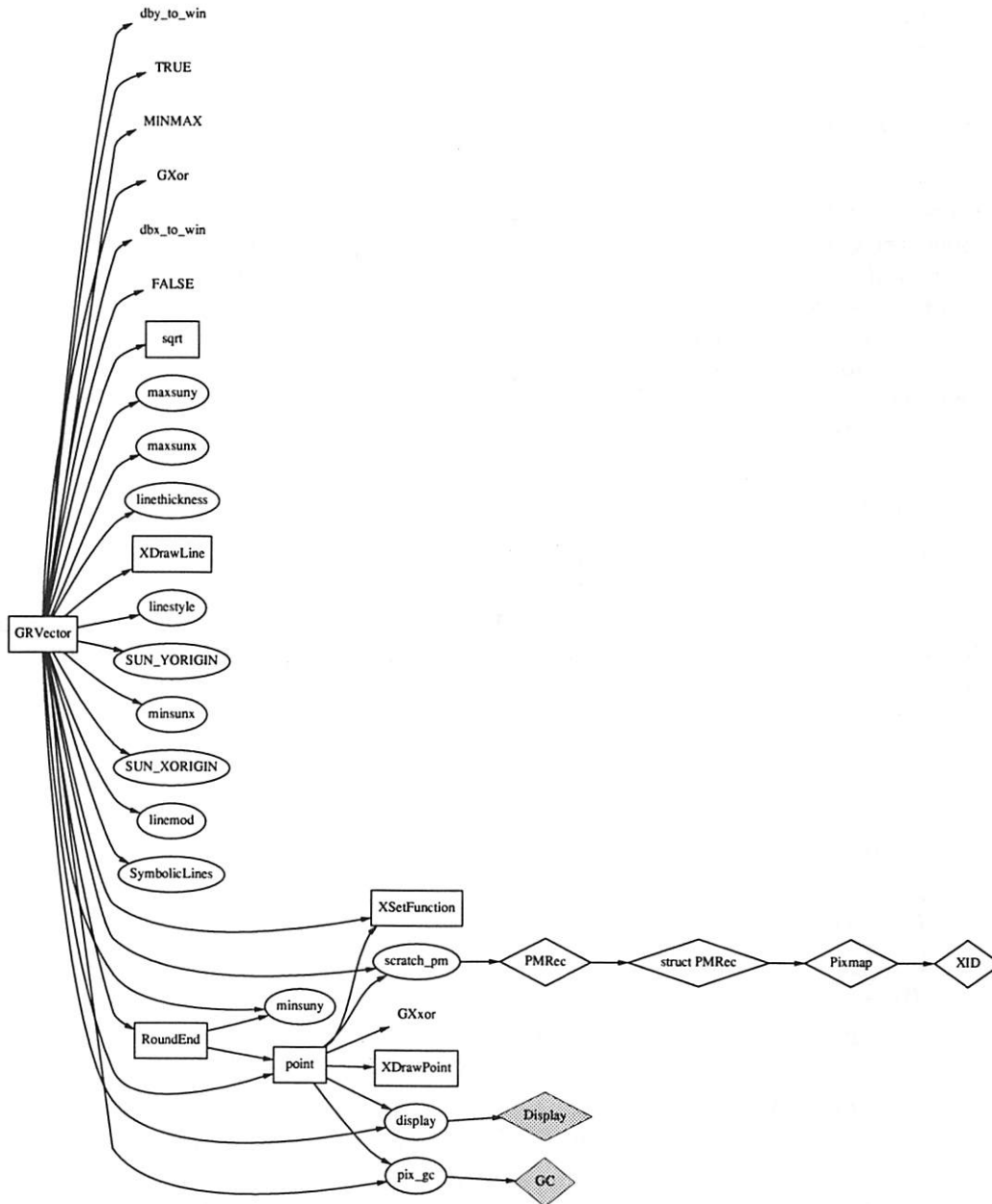


Figure 13: A Partial Subsystem Graph Rooted at the Xgremlin Function `GRVector`
(generation time: 1.86u + 3.10s = 4.96 seconds)

with the static information recorded in the CIA database.

- *Program-Graph Browser:*

Instead of serving merely as passive displays, the program graphs generated by `dagger` and `dot` can become active interfaces for interactions with the CIA program database. **Dotty**, a graph editor based on `dot` and `lefty`[13], has allowed us to build an interactive graphical interface called **ciao**⁷ for the CIA and CIA++ databases.

⁷Joint work with Eleftherios Koutsofios.

Programmers can generate textual, relational, and graphical views with great ease from pop-up menus without memorizing the cryptic dagger and subsys options. The menus are attached to program graphs and their individual nodes and edges.

6 Conclusion

By using a sequence of selective mappings through program database queries and closure operators, dagger implements a simple yet expressive mechanism to generate a large variety of interesting program graphs. The design of dagger exploits the duality between a class of entity-relationship databases and attributed directed graphs. The mapping and complexity management techniques described in this paper should be applicable to program databases for other languages – as long as they are constructed with a proper entity-relationship model. We expect program visualization tools like dagger to be widely used in the future to help programmers study, preserve, and improve structures in software systems.

7 Acknowledgements and Availability

Thanks to Susan Ashmore, David Belanger, Karen Brown, Emden Gansner, David Korn, Eleftherios Koutsofios, Steve North, and Chris Rath for their helpful comments on earlier drafts of this paper. Thanks to all those colleagues in the Software Engineering Research Department who have provided well-designed and reusable software components to make this work possible. All the dagger-related tools described in this paper can be obtained by contacting cia@mozart.att.com.

REFERENCES

- [1] Jr. Andrew D. Wolfe. Three Touches of Class. *UnixWorld*, July 1993.
- [2] Morris I. Bolsky and David G Korn. *The KornShell – Command and Programming Language*. Prentice Hall, 1988.
- [3] P. P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.
- [4] Yih-Farn Chen. The C Program Database and Its Applications. In *Proceedings of the Summer 1989 USENIX Conference*, pages 157–171, June 1989.
- [5] Yih-Farn Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, March 1990.
- [6] Yih-Farn Chen, David Rosenblum, and Kiem-Phong Vo. TestTube: A System for Selective Regression Testing. In *Proceedings of the 16th International Conference on Software Engineering*, May 1994.
- [7] Glenn Fowler. cql – A Flat File Database Query Language. In *Proceedings of the USENIX Winter 1994 Conference*, January 1994.
- [8] E. R. Gansner, S. C. North, and K. P. Vo. DAG – A Program that Draws Directed Graphs. *Software: Practice and Experience*, 18(11), November 1988.
- [9] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, March 1993.
- [10] Judith Grass. Object-Oriented Design Archaeology with CIA++. *Computing Systems*, 5(1):5–67, 1992.
- [11] Judith Grass and Y. F. Chen. The C++ Information Abtractor. In *Proceedings of the Second USENIX C++ Conference*, April 1990.

- [12] Rick Greer. Daytona User's Manual, October 1993.
- [13] Eleftherios Koutsofios and David Dobkin. Lefty: A two-view editor for technical pictures. In *Proceedings of Graphics Interface*, pages 68–76, May 1991.
- [14] Eleftherios Koutsofios and Steve North. Applications of Graph Visualization. In *Proceedings of Graphics Interface*, May 1994.
- [15] Steve Manes and Tom Murphy. C++ Development. *Unix Review*, June 1993.
- [16] Mark Opperman, Jim Thompson, and Yih-Farn Chen. A Gremlin Tutorial for the SUN Workstation. Technical Report UCB/CSD 322, Computer Science Division, University of California, Berkeley, December 1986.
- [17] Tim Parker. Pick a Pack of CASE. *Software Review*, 9(12), December 1991.
- [18] G.-C. Roman and K. C. Cox. Program Visualization: The Art of Mapping Programs to Pictures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 412–420, May 1992.
- [19] David S. Rosenblum. Towards a Method of Programming with Assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, May 1992.
- [20] Kiem-Phong Vo and Yih-Farn Chen. Incl: A Tool to Analyze Include Files. In *Proceedings of the Summer 1992 USENIX Conference*, pages 199–208, June 1992.

Developing Applications with a UIMS

Daniel Klein
LoneWolf Systems
133 Lanford Drive
Pittsburgh PA 15235-1856
dvk@lonewolf.com

A User Interface Management System (UIMS) is a collection of programs and libraries that enables an application to be partitioned into functionally distinct component parts. This partitioning recognizes that any application is comprised of three types of components, each encompassing a separate domain of expertise. These are the application domain, the display technology domain, and the user interface domain. By separating the application into dialogue utilizing a collection of agents, development time is reduced and maintainability is enhanced. Because the dialogue views the agents as black boxes, the benefits of parallel development and independent maintenance and upgrade are realized. Display technology agents can be easily reused, enabling a corporate look-and-feel. Further, an application agent can be reused with different user interfaces, creating systems tailored to different classes of users. We describe the general features of the *Alpha* UIMS, and show how a complete application can be developed using such a User Interface Management System.

1. What Is a UIMS?

Except for the simplest, non-interactive programs, every program has a user interface. The user interface is that aspect of the program that communicates with the user. This communication can be by way of a keyboard, through a point-and-click mouse-based window system, with a keypad and touch screen, a voice recognition system, or a gesturing system. The future will doubtless reveal new technologies such as eye-tracking systems or even direct computer-to-nervous system implant interfaces. Science fiction is the harbinger of the science to come, and no interface technology is too far-fetched.

In general, programs today are written with the user interface sub-domain tightly coupled to the application code. Although this is standard practice, it presents numerous problems. For example, when a bank introduces a new banking machine, it makes no difference that they already have developed bank-in-the-box programs for the three previous versions of machine. The bank must usually contract out the development of not only the new user interface, but the new banking code that resides in the machine.

The truth, though, is that the application program is very much the same for all banking machines – all that is different is the interface. In fact, the application code is in all likelihood identical for a teller's workstation, too. The underlying CPU may be different, but this is a moot point considering portable languages and current compiler technology. Yet with all the common features of the old and new versions of the banking machine, an entire new system must be developed. Even if a small change to the existing interface is made, the changes to the overall system are often quite extensive. Why?

The problem is that the interface portion of the application is very tightly coupled with the non-interface portion. The main program calls routines that access the banking-related actions and also those that access interface-related actions. Moreover, these routine calls are typically intermixed. When any change to the interface is made, the programmer responsible might conceivably have to change the entire program! If the program has been well designed and written, the interface routines are modularized. In this way, the function "get account number" is separated from the rest of the program. More often than not, however, it is the

“get keystroke” routine that is called throughout the body of code, so that the *function* of getting an account number is obscured by the *features* of getting individual keystrokes. Even if the application is properly modularized, the interface and display technology are still linked with the application domain – and it need not be!

An application that interacts with a user really consists of three things:

- 1) *The application domain code*. In the case of the banking system, this is the part of the program that understands deposits, withdrawals, transfers, and inquiries. All that it needs to know is that “customer X wants to perform transaction Y”. If the system requires personal identification numbers (PINs), the application domain may also verify the validity of a customer-id/PIN combination. The application domain does not need to know how the account number is *entered* – it only needs to know what the account number *is*. It should not care whether the account number is typed on a teller keyboard or read off of a banking card. It also does not need to know about *how* the transaction request is acquired – only *that* a withdrawal of \$100 has been requested.
- 2) *The technology-based code*. This is the part of the program that is concerned with *how* to display information on the screen, or how the buttons are read from the keypad and whether the numbers are displayed on the screen as entered, or how to read information from a magnetic card. It usually knows little or nothing about what the numbers or letters mean. It only knows that when a given button is pushed, some information must be passed to the dialogue (see below), and that some interactions with other technology-based objects may occur.

Some higher level functions may be built into the technology layer. Depending on the level of functionality desired, the technology-based code may simply expose basic objects, such as buttons and scroll bars, from which the dialogue can construct complex aggregations. It may also expose more complex objects, such as a file selection menu, which searches through a file tree interactively with the user, only returning the final target file name to the dialogue.

- 3) *The user interface code* (also called the *dialogue*). This is the part of the program that is concerned with reading an account number (via the technology-based code) and passing it to the application domain code. This may entail reading the magnetic stripe on a banking card, or accepting a typed number from a teller, although the actual mechanics are done with the cooperation of the technology-based code. The user interface code may also be concerned with processing a high-level transaction, by informing the application domain code that “customer X wants to perform transaction Y”. It does not need to know about how the transaction is processed or whether the account has sufficient cash in it. It only needs to know that a given sequence of technology-based actions signify that an application-based transaction is requested, and that an application-originated message of “give the customer \$100” means that the technology-based money dispenser should be operated accordingly.

In a system implemented with “classical” techniques, these three portions are all integrated into a single program. The application, interface, and technology code are all intermixed (often egregiously). A change in any one portion invariably affects the others. It is possible, however, to separate the three portions. Rather than develop a monolithic application that performs all three functions, a better solution divides the program along functional lines.

A user interface management system (or UIMS) is a system that encourages and enforces this division [Pfaff85, Arch92], and provides a mechanism for allowing the three distinct aspects of the application to communicate with each other. The *Alpha* UIMS is one such user interface management system.[†] The *Alpha*

[†] Other UIMS's have been developed [Foley89, Kasik89, Kolodziej87, Myers88a, Myers88b], but remain predominantly prototypes or research tools.

UIMS builds upon the experiences and ideas of the Serpent UIMS [Bass90, Hardy91, Klein91], a research prototype developed at Carnegie-Mellon University. As a development system, the *Alpha* UIMS was designed to provide a clean, powerful, and fast implementation of a UIMS, while correcting the design deficiencies discovered in the seven years since the inception of the Serpent project. The *Alpha* UIMS also adds numerous functional enhancements to the original prototype system.

A UIMS is not a “builder”, nor is it a “toolkit abstraction”. Each of these were valuable evolutionary steps in application development. A builder allows the developer to draw pictures of interaction windows. The builder then generates the code necessary to rebuild the windows using the toolkit specified. The developer must still write the interaction code necessary to use these windows, and must also interface the application. A builder provides a mechanism for speedily developing the “look” of a user interface, but not the “feel” (the interaction between objects, and between the application and the windowing system).

A toolkit abstraction is a collection of library routines which interface between an application and the underlying toolkit performing the actual display actions. As such, it removes some of the dependence on a particular toolkit, but it also provides only the “look”, and not the “feel” of an interface. Both builders and toolkit abstractions suffer from the shortcoming that the three parts of the application are still tightly coupled together. Changes to any part will often have affects in all the other parts.

A UIMS provides both the “look” and the “feel” (the presentation and interaction) of a user interface. It has all of the features of a builder, a toolkit abstraction, and more. By providing a clear separation between the three parts of an application, it also mitigates all of the problems of tightly coupling them together.

2. Implementation

An system written using the *Alpha* UIMS is implemented as a set of processes – zero or more application agents, zero or more technology agents, and exactly one dialogue manager. The dialogue manager executes the dialogue language program (written as a dependency-based collection of variables, objects with attributes, and collectors). The application agent(s) execute the user application, and the technology agent(s) execute the code needed to present interaction objects through their particular media. As far as the dialogue (and this document) is concerned, application agents and technology agents function in exactly the same way with respect to the dialogue manager. Their specific functions are different, but their mode of communication with the dialogue manager is identical. Unless a distinction needs to be made, we will refer to all technology agents and application agents simply as **agents**.

By executing the dialogue program, the dialogue manager (or **dialogue**) acts as an intermediary between all agents. No two agents ever communicate directly. Rather, an action by one agent may cause indirect interaction with another agent by direct interaction with the dialogue. Actions of an agent that are exported (that is, that are communicated to the dialogue) may cause dialogue actions that cause other actions to be communicated to other agents. However, the only path between agents is through the dialogue, and then only through explicit dialogue language actions.

Figure 1 shows the execution model of running a dialogue. The dialogue program is executed by the dialogue manager. The dialogue can interface with any number of agents (which may be either application agents or technology agents). The interface between the dialogue and the agents is maintained by the communication library subroutines, and allows the dialogue and the agents to create or delete objects and change their attributes. The agents can also invoke dialogue messages, indicating special events.

3. Nomenclature

It may be convenient to think of the instances of objects as data which is “shared” between the agent and dialogue, although there is no “shared data” in the sense of shared memory. Briefly, the notion of data shared between components of the system is a cooperative one, in which both sides of an interface keep track of commonly referenced objects of a known format. Changes made by either party are communicated

to the other party, and through continued cooperation, both parties separately maintain an up-to-date copy of the common data.

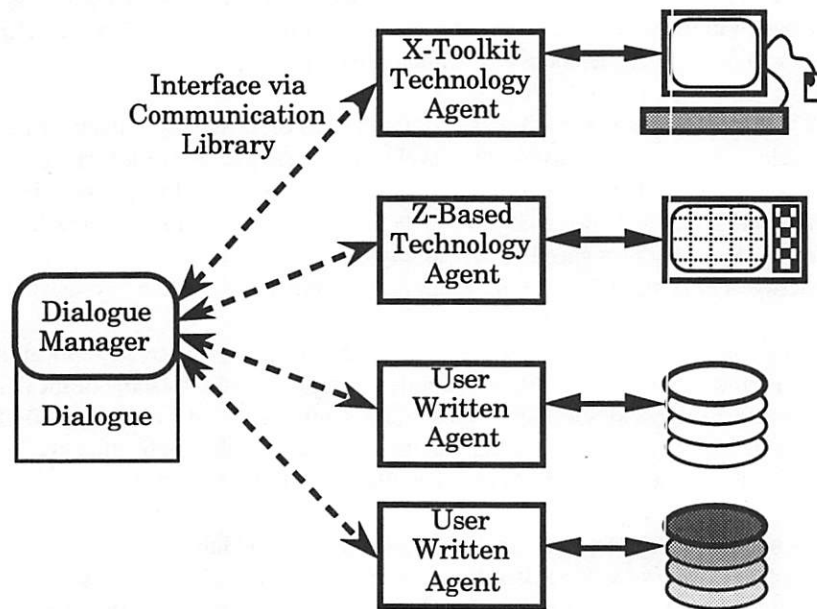


Figure 1 – UIMS Implementation

The types of objects that are “shared” between cooperating components are the objects defined in the interface definition file. These objects contain information that must be communicated between the agent and the dialogue manager. For example, in a X-based toolkit agent, an object might correspond to a push-button widget, or to a more abstract object type. In a banking agent, an object could correspond to a customer record or account. The *type* of the object is defined in an interface definition language, which creates language-specific notations to use with the communications package. The *instances* of the various types are created within the agent or dialogue programs, and the *contents* of an instance are communicated between the dialogue and agents, along with a notice that a new object instance has been created. Changes in the contents of an object instance can also be communicated, as can the deletion of an instance of a type. We use the term **object template** to describe the defining mechanism for an object. An **IDL file** (for Interface Definition Language file) describes a collection of interaction object templates. An IDL file defines the different types of objects that will be used by an agent. In general, a dialogue will use more than one IDL file, providing access to the templates for the objects associated with multiple agents.

The term **object** describes an instance of a template. There may of course be multiple objects instantiated from a single template. Objects are created statically or dynamically, depending on the agents actions.

An **attribute** is a component of an object. Objects are aggregate structures, with attributes containing a string, an integer, a floating point number, etc. Attributes can also be defined to be array of elements of a single type. A **message** is a special attribute of an object. It is used to signal the dialogue that a special condition exists within the application. A message differs from ordinary data in that invoking a message signals an *event*, whereas all other changes in an object’s attributes are *data* changes. When the agent invokes a message for an object, a special code snippet associated with the object can be executed.

A **collector** is a mechanism for grouping objects and variables according to a common existence condition. Collectors can exist solely based on a runtime condition, or they can be “bound” to another object. In this way, the dialogue can create a group of objects when another object is created. For example, a collection of buttons and labels can be created when a new customer object is created. A **stanza** is a special type of collector, bound to an instance of dialogue-specific data.

The **interface library** is a collection of C (or other language) routines which enables agents and the dialogue to communicate. Agent developers use the interface library explicitly to register or respond to the creation, destruction, or change to an object. The dialogue uses the interface transparently, so that changes to objects are reflected by the automatic execution of dependency-based and procedural code snippets.

The **dialogue language** is a programming language provided as part of the *Alpha* UIMS. It is one of the key elements of the *Alpha* UIMS, and provides features including typeless variables, dependency-based and procedural code elements, an automatic relational database, easy access to dynamically created interaction objects, collectors, and an interface to other programming languages, such as C. A **dialogue** is a program written in the dialogue language. When a dialogue executes, it uses user-written and system-supplied agents to create an entire system, communicating with the interface library. Unlike most other programming languages, a dialogue is a *reactive* program (or in other words, as an event-based system, where events are defined in terms of creation of, changes to, and destruction of interaction objects). A **snippet** is a portion of a dialogue program that is executed in response to a state change in the variables or object instances in the dialogue. Rather than initiating actions, a dialogue primarily responds to events in the application programs. Essentially, the dialogue is the communications manager between the agents, passing, changing, and regulating data that is shared between them. The dialogue also enables the user agents to interface with the various display technologies while separating the concerns of all the components of the system.

4. Pictorial Overview

The *Alpha* UIMS is divided into four conceptual components, shown in Figure 2, and explained in the list below it. Note that this figure is merely a simplified rendition of Figure 1.

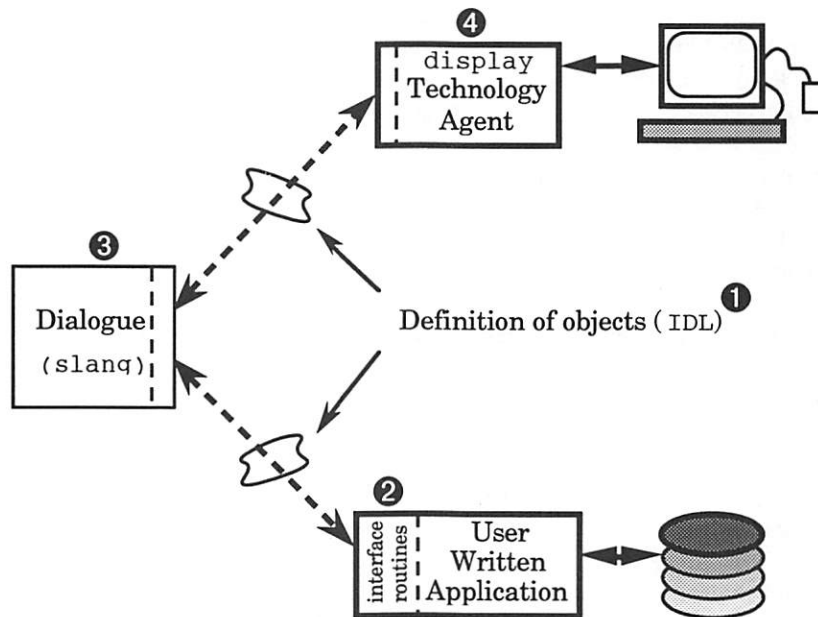


Figure 2 – Parts of a UIMS

- 1) The interface definition language (IDL), which is used to define object templates that are communicated between the dialogue and the various agents.
- 2) The interface library routines, which are used to effect communications between user-written agents and the dialogue. The interface routines are also transparently used by the dialogue.

- 3) The dialogue language, which is used to describe the layout and function of the dialogue, using the user-written applications and/or the *Alpha* standard technology agents.
- 4) Display technology agents. Although the *Alpha* user can create their own technology agents, at least one is supplied with the *Alpha* UIMS.

5. Building a System with the Alpha UIMS

To develop a system using the *Alpha* UIMS, the first thing that must be done is to decide how the system will be partitioned, and what objects will be supported by the various agents. The templates for the objects are defined using IDL.

Once the IDL file has been created, the dialogue and the agent(s) can then be written. The dialogue uses aggregate object notation to reference the IDL object types, while the agent uses the interface library routines. The dialogue can also reference existing agents, such as the `display` technology agent.

5.1. Partitioning of Control

One question that must be asked when developing a system using the *Alpha* UIMS is “where does the responsibility for control reside?” The reason for this question is that there is never a sharply drawn line between the responsibility of the dialogue and an agent – the line is flexible, depending on your design decisions. Here are two examples of this fuzzy line:

- 1) Consider the processing of a shipping address in an order-entry system. If the application agent has an “address” object, the dialogue can pass each part (street name, city, state, Zip code, etc.) to the agent as it is filled in by the user. The agent must then perform tests to determine if each part of the address is legal (and reject, for example, a four-digit Zip code) – the dialogue is then merely a data pass-through between the technology agent and the application agent.

An alternative implementation is one in which the dialogue performs all of the validation tests itself, and only gives the application agent a fully filled-in address object. Either approach is a valid use of the *Alpha* UIMS. In fact, there are numerous intermediate implementations, where the dialogue and the application agent share the validation responsibilities.

- 2) Consider a technology agent, the program responsible for drawing on the screen. If the agent provides a mechanism for displaying a collection of “radio buttons,” this can be done in many ways. One way is to expose the underlying toolkit, so that Motif objects of type `XmRowColumn` and `XmToggleButton` are exposed, along with the attributes necessary to place the buttons on a pleasing layout (such as `XmNradioBehavior`, `XmNradioAlwaysOne`, `XmNOrientation`, etc.). This means that the toolkit agent is simply a Motif interface, and the dialogue programmer must be intimately familiar with the details of Motif.

An alternative implementation is one in which the agent knows all of the details of Motif (if indeed Motif is used at all!), and simply provides a canvas onto which exclusive buttons can be drawn. The toolkit agent worries about the specific toolkit calls. All the dialogue programmer does is say button #1 is to the right of button #2, etc. Each approach is “valid”, although the *Alpha* UIMS distribution takes the latter approach, and attempts to relieve the dialogue programmer of the odious task of becoming familiar with Motif.

6. The Interface Library

An agent programmer uses the interface library routines to communicate with the dialogue. Interface library functions fall into five categories:

- 1) Interface initialization and shutdown.
- 2) Object creation and destruction.
- 3) Object modification, including changing object attributes, and sending object messages.
- 4) Transport and status (bundling collections of actions into optimized packages for transport across the interface, targeting of calls, status messages, etc.).
- 5) Receipt and processing of changes from the dialogue.

The C language interface provides routines to call from a C-based agent – other language interfaces are also available. The file `time_of_day.c` in the appendix shows how the interface library is used.

7. The Dialogue Language (Slang)

The dialogue language (called *slang*) is used to describe a dialogue in the *Alpha* UIMS. A dialogue is defined as the interaction between the various agents which comprise a system. Typically, there will be a technology agent interacting with the user via a terminal, keyboard, mouse, etc., and an application agent doing the data manipulations in the application domain. The dialogue moves information between the agents, possibly modifying it *en route*, and possibly also making decisions as to when information will be moved. In this way, a dialogue interacts with interface objects defined by the various agents that it uses to provide a flexible interface to the user.

7.1. Mutable Types

The dialogue language does not require that variables be declared to be of a specific type. The type of a variable “mutates” according to the type and value of the operation which assigns it. Variable may be of type:

```
boolean integer real string id bundle
```

Once a variable has been assigned a value and a type, the type remains unchanged until a new value and type are assigned to it. Some type conversions are automatic. For example, an integer expression may be used in a conditional statement – the integer value of the expression is converted to boolean before the conditional statement is evaluated. Special routines are also provided to copy the value of a variable and convert that copy to a specific type.

7.2. Dependency-Based and Procedural Code Snippets

A code snippet is a small piece of code that is executed given a set of conditions. The dialogue language supports two different types of code snippets – dependency-based and procedural.

Dependency-based snippets are executed when a specific *relationship condition* is updated. These relationships are defined to be changes in an independent portion of an expression. This is similar to the model provided by relational databases.

Procedural snippets are executed when a specific *trigger event* occurs. External triggers are typically object methods or creation actions. This is similar to the model used by most programming languages.

Dependency-based code is something which is unfamiliar to many programmers. However, the notion of data-dependency *is* something that is familiar. Any program which does something meaningful is predicated on the fact that things change. When changes occur, other things change in response (sometimes

moderated by thresholds). Dependency-based programming expresses these relationships and cascades of data in a simple, straightforward manner.

7.2.1. C Example

Almost all high-level programming languages enable you to create data dependencies manually. Typically, you accomplish this by writing code that essentially states “this variable has changed, therefore I must also change the following”. The nature of most languages, however, is such that however you state the dependency, you will make these statements *often*, and usually *redundantly*. Consider this simple example:

```
average = (left + right) / 2;
while (1) {
    /* main body of code goes here */
    if (changed (&left))
        average = (left + right) / 2;
    if (changed (&right))
        average = (left + right) / 2;
}
```

In this example, we define a variable called `average` to be the average of `left` and `right`. Whenever either of the independent variables (`left` or `right`) change, we manually update `average`. In this way, data dependencies are expressed by asking the question “who depends on me?” The problem is that adding a dependency to a variable requires that you change the code associated with *all* of the independent variables in the dependency. If we want to change the definition of `average` to include a third variable called `center`, we have to update the code in four separate places to look like this:

```
average = (left + right + center) / 3;
while (1) {
    /* main body of code goes here */
    if (changed (&left))
        average = (left + right + center) / 3;
    if (changed (&right))
        average = (left + right + center) / 3;
    if (changed (&center))
        average = (left + right + center) / 3;
}
```

This redundant updating of code can become very cumbersome and error-prone as the number of dependencies grows beyond that shown in this simple example.

7.2.2. Slang Example

A dependency-based programming language expresses data dependencies from the opposite viewpoint, by asking the question “on whom do I depend?” Dependency-base snippets define the data dependencies, which are reevaluated automatically whenever an independent variable changes. The first simple example we saw above would be written in Slang as:

```
average : (left + right) / 2;
```

This snippet *defines* `average` to be the average of `left` and `right`. Whenever either of the independent variables change, `average` is updated automatically. If we want to update the definition to include the third variable `center`, we simply change the code snippet to be:

```
average : (left + right + center) / 3;
```

Because each data dependency is stated in terms of what it depends on (instead of what depends on it), changing data dependencies is greatly simplified, and is substantially less error-prone than in conventional programming languages.

7.3. Independent Variables

Essentially, an independent variable is anything that could be considered a “right-hand-side” of an expression (or in C, an rhs). Because a dependency-based snippet may contain more than one statement (and each statement may contain complex expressions), a snippet may be dependent on more than one independent variable. For example, this snippet is dependent on both `alpha` and `beta`:

```
foo : { foo = alpha + beta }
```

If either `alpha` or `beta` changes, the snippet is automatically reevaluated. If a snippet only has a single statement, and that statement assigns the snippet “tag”, it may be written using an abbreviated notation. The previous snippet can be abbreviated as:

```
foo : alpha + beta;
```

This abbreviated notation is the one most often used when writing a dialogue.

This next snippet is also dependent on both `alpha` and `beta`, although this snippet has two statements instead of one. If *either* `alpha` or `beta` changes, the *entire* snippet (both assignments) is reexecuted.

```
foo : {  
    foo = alpha;  
    baz = beta;  
}
```

8. An Example Dialogue

Figure 3 shows how the dialogue language can be used to construct a dialogue using a single agent. In this case, the agent is a technology based agent called `display`. This example is much simpler than an “real” application, and merely illustrates some of the features the dialogue language. A more complete example consisting of a dialogue, an application agent, and a display agent can be found in the Appendix section of this paper.

Regardless of this example’s complexity, it is worth noting that it takes less than two minutes to develop, and 0.2 seconds to compile (with no link phase needed!). A similar example developed using any GUI builder would take much longer to assemble, and regardless of the development method used, compilation and linking using X windows is always very time consuming. An equivalent example written using Motif takes 20 seconds to compile and link. Similar ratios are seen for larger examples.

Although compilation time may not be the largest factor in a development effort, fast compiler turnaround reduces developer frustration, and increases productivity. A fast compiler and realization engine also encourages the exploration of alternatives, leading to a better final product.

In this example, the dialogue declares a single variable called `counter`, and uses the `display` agent to create four objects (one `canvas`, one `label`, and two `buttons`). The placement of the buttons and labels are specified in absolute coordinates, although the Appendix section shows more complex placement schemes involving relative placement with optional offsets.

The `text` attribute of the `show_it` button is defined to be dependent on the variable `counter`. This means that whenever the value of `counter` changes (in this example, by pushing the `click` button), the value of `show_it.text` is also changed. Thus, every time the `click` button is pushed, the `show_it`

label is incremented by one. This dependency-based specification enables the dialogue writer to specify arbitrarily complex interrelationships between agents and attributes. The fact that the `text` attribute is a string and the value of the counter is irrelevant – the dialogue performs automatic type conversion.

```

use display;

VARIABLES:
    counter : 0;

AGGREGATES:
draw_here : canvas {
    height : 300;
    width  : 150;
}

show_it : label {
    parent : draw_here;
    x      : 50;
    y      : 50;
    text   : counter;
}

click : button {
    parent : draw_here;
    x      : 50;
    y      : 150;
    text   : "Push";
    pushed : begin counter += 1 end
}

quit : button {
    parent : draw_here;
    x      : 50;
    y      : 200;
    text   : "Quit";
    pushed : begin exit () end
}

```

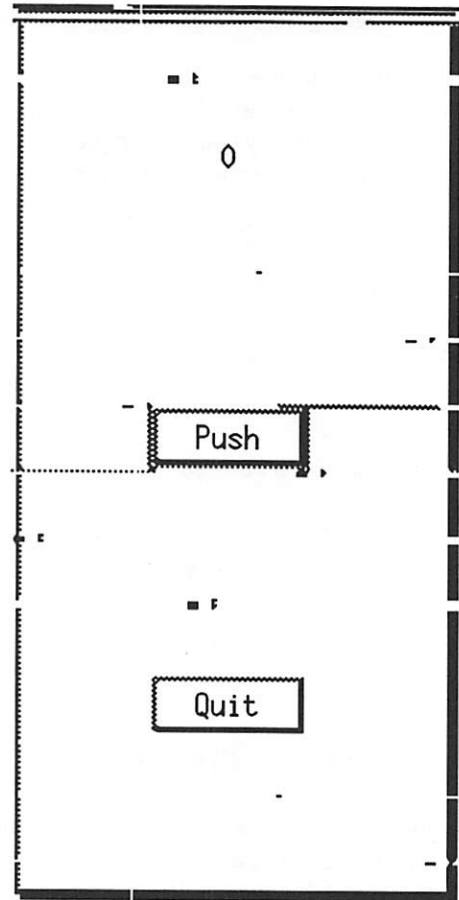


Figure 3 – A simple dialogue example

9. Conclusions

Using a UIMS can greatly speed application development and enhance maintainability. This is because the components of the systems – application, dialogue, and technology – are separated from each other. This means that:

- 1) Each component can be developed independently. Parallel development is faster than serial.
- 2) Each component can be developed by a person expert in that particular field. Graphics experts can develop technology agents, human factors experts can develop the dialogue, and application experts can develop the application agent.
- 3) Changes to one component of the system do not affect other components. The application agent may be changed independently of the dialogue, and the technology agent may be made to support new objects without changing the dialogue. This separation greatly reduces maintenance costs.
- 4) Overall development time on any component is greatly reduced. The dialogue compiler is exceptionally fast, so many alternative dialogues can be tested before the final interface is selected.

Errata for "Developing Applications with a UIMS"

Figures 3, 4, and 5 were damaged during printing of the proceedings. Corrected versions of these three graphs are presented here. Please make a note in the printed proceedings and include this sheet near the relevant pages.

```
use display;

VARIABLES:
  counter : 0;

AGGREGATES:
draw_here : canvas {
  height : 300;
  width  : 150;
}

show_it : label {
  parent : draw_here;
  x      : 50;
  y      : 50;
  text   : counter;
}

click : button {
  parent : draw_here;
  x      : 50;
  y      : 150;
  text   : "Push";
  pushed : begin counter += 1 end
}

quit : button {
  parent : draw_here;
  x      : 50;
  y      : 200;
  text   : "Quit";
  pushed : begin exit () end
}
```

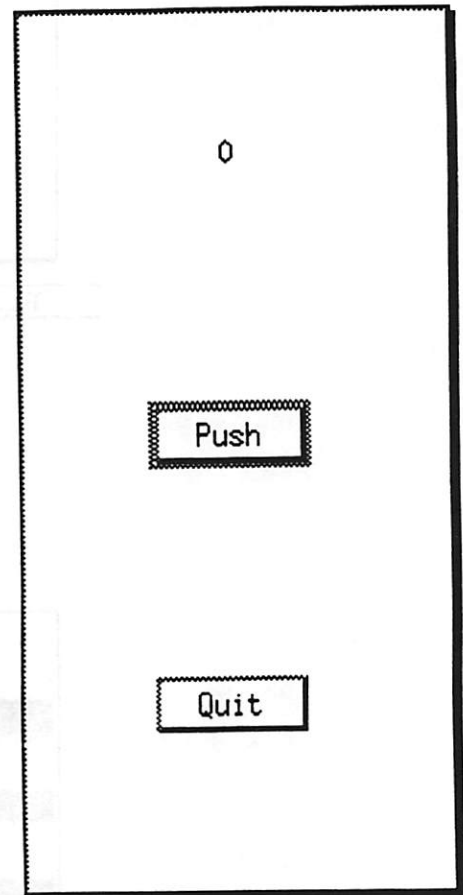


Figure 3 - A simple dialogue example

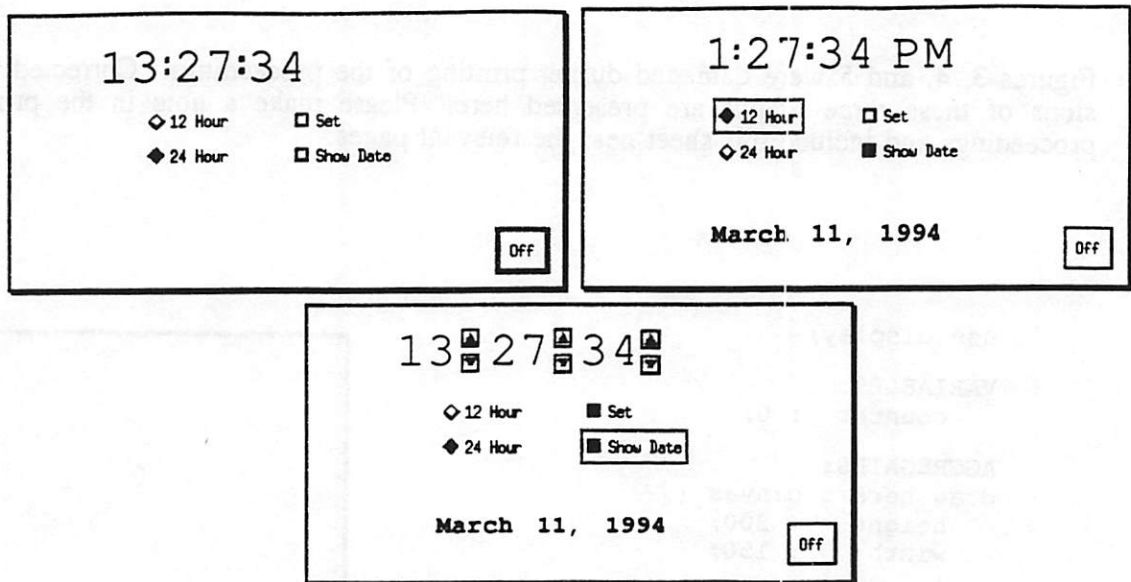


Figure 4 - Various appearances of clock1.sl

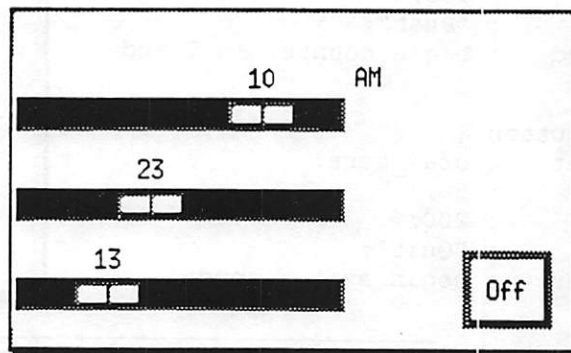


Figure 5 - Appearance of clock2.sl

- 5) Components can be reused. For example, one technology agent can be used with many systems. If different interfaces are desired, multiple dialogues can be used without changing the application agents. A single dialogue can also be told to use different technology agents, so a dialogue can display to X windows or Windows NT.
- 6) Upgrades to new versions of technologies are easier. When a new release of a toolkit is purchased, only the technology agents need to be recompiled (and not every dialogue using it). If one technology agent is shared (as is usual), once that agent is upgraded, *all* the dialogues using it are upgraded without recompilation or relinking.
- 7) The UIMS is inherently distributable, and can run on a heterogeneous network. Each of the components can run on a different machine, communicating over a network.

Appendix – Using a UIMS to Build Two Interfaces to One Application

Because the dialogue, technology, and application are separate, the UIMS easily allows an application to have multiple interfaces. This appendix section shows a simple application (a clock) with two radically different interfaces. The main components of the system (the interface definition, the dialogue, and the application agent) are shown.

A.1. clock.idl

This is the interface definition for the objects used by the clock agent. The executable image is defined to be `time_of_day`, and a single object type is defined here, namely a `daytime` object. The dialogue or the application agent may create instance of this object.

```
application objects <<"time_of_day">>

daytime {
    hrs      : integer;
    mins     : integer;
    secs     : integer;
    date     : string;
}
```

A.2. time_of_day.c

This is the `time_of_day` agent for the clock interfaces. Once the interface definition (in `clock.idl`) has been compiled, it creates the C language definitions of the objects, which can then be used by the agent and the dialogue.

Note that the agent does not actually change the time of day – it simply maintains an internal delta. Consequently, the agent is much more complex than really it needs to be. It would be simpler to just change the time of day with `settimeofday`, but we don't assume root privileges.

```
#include "interface.h"                /* The interface library */
#include "clock.h"                    /* The output from clock.idl */
#include <time.h>
#include <stdlib.h>

static daytime_obj *delta;            /* daytime_obj comes from clock.h */
static daytime_obj last_time;
```

```

static void find_time (daytime_obj *dto)
{
    time_t when;
    struct tm *now;

    when = time ((time_t *)0);
    now = localtime (&when);
    dto->hrs = now->tm_hour;
    dto->mins = now->tm_min;
    dto->secs = now->tm_sec;
    dto->date = malloc (20);
    (void) strftime (dto->date, "%B %e, %Y", now);
}

static void clock_create (xdl_idx obj_num, void *obj)
{
    delta = obj;
}

static void adjust_time (daytime_obj *dto)
{
    int hrs, mins, secs;
    /* If the delta hasn't been created yet, no adjustment needed */
    if (delta == (daytime_obj *)0)
        return;

    get_attr (&secs, delta, daytime$secs);
    if (dto->secs + secs < 0) {
        secs += 60;
        set_attr (delta, daytime$secs, secs);
    }
    else if (dto->secs + secs >= 60) {
        secs -= 60;
        set_attr (delta, daytime$secs, secs);
    }
    dto->secs += secs;

    get_attr (&mins, delta, daytime$mins);
    if (dto->mins + mins < 0) {
        mins += 60;
        set_attr (delta, daytime$mins, mins);
    }
    else if (dto->mins + mins >= 60) {
        mins -= 60;
        set_attr (delta, daytime$mins, mins);
    }
    dto->mins += mins;

    get_attr (&hrs, delta, daytime$hrs);
    if (dto->hrs + hrs < 0) {
        hrs += 24;
        set_attr (delta, daytime$hrs, hrs);
    }
    else if (dto->hrs + hrs >= 24) {
        hrs -= 24;
        set_attr (delta, daytime$hrs, hrs);
    }
    dto->hrs += hrs;
}

```

```

static void done (void)
{
    exit (0);
}

main (int argc, char **argv)
{
    daytime_obj *dlg_time;
    /*
     * This is a trivial router - it only processes create, modify and
     * dialogue shutdown actions.
     */
    static router_t daytime_router = {
        (a_none)0, clock_create, (a_modify_attr)0, (a_modify_row)0,
        (a_modify_idx)0, (a_destroy_object) 0, (a_begin_group) 0,
        (a_end_group) 0, done
    };

    Alpha_init ("clock.xdl");
    register_type_router (clock$daytime, &daytime_router);
    /* Create the daytime object for the interface to use */
    begin_group ();
    dlg_time = create_object (clock$daytime);
    find_time (&last_time);
    *dlg_time = last_time;                      /* Copy whole object */
    changed_object (dlg_time);
    end_group ();

    while (1) {
        sleep (100*MS);
        find_time (&last_time);                /* Get time of day */
        while (change_available ()) /* See if the user has changes */
            process_next_change (TRUE);
        adjust_time (&last_time);              /* Adjust according to delta */
        begin_group ();
        *dlg_time = last_time;                  /* Copy whole object */
        changed_object (dlg_time);              /* Ship changes to interface */
        end_group ();
    }
}

```

A.3. clock1.sl

This dialogue creates an interface for a digital clock. The technology agent is called `display` (for the sake of brevity, it is not shown here). The time of day and date fields come from the application agent called `clock` (the name of the executable image is defined in `clock.idl`). The dialogue also accepts changes in the time of day via push-buttons, and sends those changes to the `clock` agent. Figure 4 shows the way the clock appears as various controls are interacted with.

Note that the hours, minutes, and seconds fields shown using the `display` agent are dependent on the values managed by the `clock` agent. Whenever the `clock` agent changes the values of the `daytime` object, the dialogue automatically instructs the `display` agent to make the corresponding changes on the screen. The `clock` agent has no idea how the date and time are being displayed (if at all!). Similarly, the dialogue does not know how the `display` agent actually gets the bits on the screen, nor how the `clock` agent determines the time (or how often). The dialogue simply expresses the interdependencies between objects in potentially disparate sections of the application, and routes information between them. In this way, the component parts are much simpler than if they knew details about the other parts of the system.

There are a number of specific features to note in the `clock1` dialogue (look for the cross reference numbers in the code):

- ❶ Two agents. In this example, we use a technology agent (`display`) to draw things on the screen and an application agent (`clock`) to maintain the time of day.
- ❷ Abstract object templates. An `xlabel` is the same type as a `label`, except the `parent` attribute is set to `drawing_area` (unless otherwise overridden). This enables dialogues to be written with less typing, and allows abstract objects to be created. These abstract objects can be substantially more complex (see item ❸).
- ❸ Font and appearance objects. Because many objects can share the same appearance, there is no reason to force the redeclaration of these appearances for each object. An appearance object allows many objects to have the same appearance, simply by referencing it. If the appearance object is changed, all objects which reference it also change in appearance (so all objects in a group can be made to change color or font by simply changing a single appearance object).
- ❹ Relative placement. An object can have its position dependent on the size of other objects – if the `drawing_area` were to change size, so would the position of the `quit` button. See item ❷ for another type of relative placement.
- ❺ Collector. A collector is a way of grouping a set of objects with a common existence condition. In this case, the existence is bound to the creation of a `daytime` object by the `clock` agent. When an instance of the object is created, an instance of the collector is created (which in turn causes all of the buttons and labels to be created). Another example of a collector is shown in item ❹.
- ❻ A more complex object abstraction – in this case the height and width, sensitivity and parent attributes are all set. Note that this means that the “colon” characters separating the hours, minutes and seconds are really tiny arrow buttons which are insensitive. When the `set_time` button is pushed in, the arrows grow and become sensitive, allowing the user to change the time.
- ❼ Relative placement. Although an object’s placement can be specified in absolute coordinates, an object can also be said to be across or down from another object. In this case, the `hrs_down` button is to the right of the hours field and down from the `hrs_up` button. The `hrs_up` object (above) also adds a relative `y` offset of 7 units if the `set` attribute of the `set_time` button is off.
- ❽ Conditional formatting. The `seconds` label will also have “AM” or “PM” appended to it if we are in 12 hour mode (determined by the state of the `civilian` button).
- ❾ Another collector. This collector has as its existence condition the value of the `set` attribute of the `show_date` toggle button. If the button is “on”, the label called `today` appears. If it is “off”, the button disappears.

```
use display;  
use clock;
```

❶

```
AGGREGATES :
```

```
drawing_area : canvas {  
    height : 200;  
    width : 400;  
}
```

```

xlabel => label {
    parent => drawing_area;
}

cour20 : font {
    family : "courier";
    weight : "bold";
    slant : "r";
    pixel_size : 20;
}

c20 : appearance {
    font : times20;
}

cour34 : font {
    family : "courier";
    weight : "bold";
    slant : "r";
    pixel_size : 34;
}

cb34 : appearance {
    font : cour34;
}

quit : button {
    parent : drawing_area;
    text : "Off";
    height : 35;
    width : 40;
    x : drawing_area.width - quit.width - 10;
    y : drawing_area.height - quit.height - 10;
    pushed : begin exit () end
}

all_the_stuff : collector bound_to (daytime) {
    AGGREGATES:
    one_of_these : exclusive_canvas {
        parent : drawing_area;
        height : 100;
        width : 200;
        x : 95;
        y : 65;
    }

    civilian : exclusive_button {
        parent : one_of_these;
        text : "12 Hour";
    }

    military : exclusive_button {
        parent : one_of_these;
        down : civilian;
        text : "24 Hour";
        set : true;
    }
}

```

```

set_show : toggle_canvas {
    parent : drawing_area;
    x : 200;
    y : 65;
}

Set_Time : toggle_button {
    parent : set_show;
    text : "Set";
}

Show_Date : toggle_button {
    parent : set_show;
    down : Set_Time;
    text : "Show Date";
}

delta : daytime { /* Initialized to zero */ }

clock_area : canvas {
    parent : drawing_area;
    x : 75;
    y : 15;
}

clabel => label {
    parent => clock_area;
}

carrow_button => arrow_button {
    parent => clock_area;
    sensitive => Set_Time.set;
    height => Set_Time.set ? 18 : 8;
    width => Set_Time.set ? 18 : 8;
}

hours : clabel {
    appearance : cb34;
    text : civilian.set ?
        daytime.hrs > 12 ?
            int_to_str (daytime.hrs - 12, "%02d") :
            int_to_str (daytime.hrs, "%02d") :
            int_to_str (daytime.hrs, "%02d");
}

hrs_up : carrow_button {
    direction : arrow_up;
    y : Set_Time.set ? 0 : 7;
    across : hours;
    pushed : begin delta.hrs += 1; end
}

hrs_down : carrow_button {
    direction : arrow_down;
    across : hours;
    down : hrs_up;
    pushed : begin delta.hrs -= 1; end
}

```

6

7

```

minutes : clabel {
    appearance : cb34;
    text : int_to_str (daytime.mins, "%02d");
    across : hrs_up;
}

mins_up : carrow_button {
    direction : arrow_up;
    y : Set_Time.set ? 0 : 7;
    across : minutes;
    pushed : begin delta.mins += 1; end
}

mins_down : carrow_button {
    direction : arrow_down;
    across : minutes;
    down : mins_up;
    pushed : begin delta.mins -= 1; end
}

seconds : clabel {
    appearance : cb34;
    text : int_to_str(daytime.secs, "%02d") @
        (civilian.set ? (daytime.hrs < 12 ? " AM" : " PM") : "");
    across : mins_up;
}

secs_set : collector exists if (Set_Time.set) endif {
    AGGREGATES:
    secs_up : carrow_button {
        direction : arrow_up;
        across : seconds;
        pushed : begin delta.secs += 1; end
    }

    secs_down : carrow_button {
        direction : arrow_down;
        across : seconds;
        down : secs_up;
        pushed : begin delta.secs -= 1; end
    }
}

todays_date : collector exists if Show_Date.set endif {
    AGGREGATES:
    today : xlabel {
        appearance : c20;
        text : daytime.date;
        x : 85;
        y : 130;
    }
}
}

```

8

9

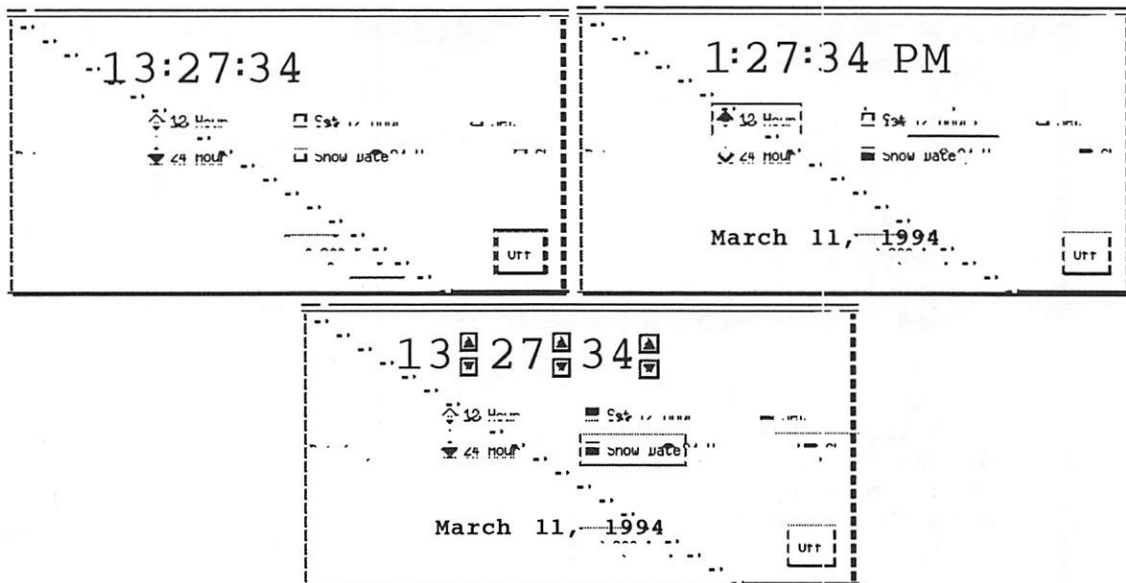


Figure 4 – Various appearances of clock1.sl

A.4. clock2.sl

This shows a different dialogue (or interface) for the same two agents as in the previous example. Neither agent (display or clock) has been changed (or recompiled, or even relinked!). Only the dialogue is different, giving a different look-and-feel to the same application.

In this second example, the clock is displayed as sliders – the time is changed by moving the slider bar, and the user does not see the date. Figure 5 shows the new interface defined with clock2.sl. This second example demonstrates that one application can be given multiple interfaces, depending on the functions the designer wishes to expose to the user. These differences in interaction can be completely invisible to the application agent.

```

use display;
use clock;

AGGREGATES :

drawing_area : canvas {
    height : 150;
    width : 250;
}

quit : button {
    parent : drawing_area;
    text : "Off";
    height : 35;
    width : 40;
    x : drawing_area.width - quit.width - 10;
    y : drawing_area.height - quit.height - 10;
    pushed : begin exit () end
}

```

```

all_the_stuff : collector bound_to (daytime) {
  AGGREGATES:

  delta : daytime { /* Initialized to zero */ }

  hours : slider {
    parent : drawing_area;
    y : 20;
    minimum : 1;
    maximum : 12;
    value : daytime.hrs mod 12 == 0 ? 12 : daytime.hrs mod 12;
    orientation : LeftToRight;
    changed : begin delta.hrs += hours.value - daytime.hrs; end
  }

  meridian : label {
    parent : drawing_area;
    across : hours;
    y : 20;
    text : daytime.hrs < 12 ? "AM" : "PM";
  }

  minutes : slider {
    parent : drawing_area;
    down : hours;
    maximum : 60;
    value : daytime.mins;
    orientation : LeftToRight;
    changed : begin delta.mins += minutes.value - daytime.mins; end
  }

  seconds : slider {
    parent : drawing_area;
    down : minutes;
    maximum : 60;
    value : daytime.secs;
    orientation : LeftToRight;
    changed : begin delta.secs += seconds.value - daytime.secs; end
  }
}

```



Figure 5 – Appearance of clock2.sl

Bibliography

- [Arch92] The UIMS Tool Developers Workshop, "A Metamodel for the Runtime Architecture of an Interactive System," *SIGCHI Bulletin*, 24:1(32) Jan. 1992
- [Bass91] Bass, Len; Joëlle Coutaz; *Developing Software for the User Interface*, Addison Wesley Publishing Company, 1991, ISBN 0-201-51046-4
- [Bass90] Bass, Len, Brian Clapper, Erik Hardy, Rick Kazman, Robert Seacord; "Serpent: A User Interface Environment," *Proceedings, 1990 USENIX Winter Conference*, Jan. 1990
- [Boehm88] Boehm, Barry W.; "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, 21:5, May 1988
- [Fisher87] Fisher, Gary E.; *Application Software Prototyping and Fourth Generation Languages*, National Bureau of Standards, May 1987
- [Foley89] Foley, James, et al.; "Defining Interfaces at a High Level of Abstraction," *IEEE Software*, Jan. 1989
- [Hardy91] Hardy, Erik J., Daniel V. Klein; "The Serpent UIMS", *Proceedings of the European Unix User's Group*, Nice FRANCE, Oct. 1990
- [Kasik89] Kasik, David J., et al.; "Reflections on Using a UIMS for Complex Applications," *IEEE Software*, Jan. 1989
- [Klein91] Klein, D.V.; *Serpent: System Guide*, Software Engineering Institute User's Guide CMU/SEI-91-UG-2, April 1991
- [Kolodziej87] Kolodziej, Stan; "User Interface Management Systems," *Computerworld*, July 8, 1987
- [Myers88a] Myers, Brad A.; *Tools for Creating User Interfaces: an Introduction and Survey*, Carnegie Mellon University, 1988, CMU-CS-88-107
- [Myers88b] Myers, Brad A.; *The Garnet User Interface Development Environment: a Proposal*, Carnegie Mellon University, 1988, CMU-CS-88-153
- [Pfaff85] Pfaff, G. (Ed.); *Seeheim Workshop on User Interface Management Systems*, Springer-Verlag, Berlin, 1985

SPP - Low Tech, Practical, UNIX Software Portability

John Sellens – jmsellens@uwaterloo.ca

University of Waterloo, Waterloo, Ontario, N2L 3G1 Canada

Abstract

The great number of UNIX variants and the huge amount of good, freely available software makes simple software portability a very worthwhile and useful goal. Portability has traditionally been obtained with software package specific solutions, configuration scripts, `#ifdef`'s and so on. This paper describes SPP, a small collection of include files, library routines and utilities that can be installed once and utilized by many, different software packages. SPP does not require that a software author change his or her programming style or techniques, and frees the author from worrying about the variations among machines, allowing him or her to concentrate on the program instead of the operating system.

1 The Motivation for SPP

At the University of Waterloo, most UNIX software support is provided by central support groups, and much effort is expended to provide a consistent set of software and tools across all supported platforms[1]. Automated tools are used to build and distribute software to multiple hardware/operating system combinations (16 at last count) from a single source repository. The source must be buildable without human intervention, and any source configuration or option selection must be done automatically.

Hundreds of megabytes of source is used to build hundreds of megabytes of installed software, much of it obtained over the network, or from other freely-distributable sources. Much of this software is excellent, and many packages come with extensive configuration files and scripts, that work hard to make software installation as painless as possible¹.

If portability to multiple architectures is desired (and it usually is), software authors must go to considerable effort to construct configuration scripts, or resort to `Makefile` editing and `cpp #ifdef`'s to achieve a measure of portability. The amount of effort required to construct "portable" software can sometimes discourage software authors from distributing their software. In addition, the portability scheme adopted can sometimes make it more difficult to port it to another architecture².

There does not seem to be a common set of tools that a software author can easily use to write portable software. Each author must make the effort to make the software portable, and must learn enough about enough systems to do it properly. Much effort is expended (and duplicated) in the goal of portability (or even transportability). If software can be written by the original author in a way that anticipates portability, and using a relatively common set of tools, then everyone will benefit³. And, with an appropriate set of tools, an author can create software that is far more likely to build and install on platforms that the author has no knowledge of.

SPP is intended to provide a small set of tools that are a practical approach to portability. The tool set must be small and simple enough that it won't be a burden to install and use, and it must be easy to understand how it works, and what it will do (e.g. the output of `make install` must be understandable). SPP can only be

¹The most notable is probably Larry Wall's *Configure* script; the version distributed with *perl* 4.0 is over 4,000 lines long, and can generate 500 lines of output! *Configure* was, however, generated by another program, *metaconfig*.

²Consider the case of code liberally sprinkled with "`#ifdef vax`" to determine byte-ordering.

³This is, after all, part of the "UNIX philosophy".

useful if it is widely installed and available; more software will be more easily installable if the author can rely on the availability of a useful portability tool set.

The name, SPP, is intended to reflect this reality, and these goals. The name is an abbreviation for "Standard Portability Package", though the "S" is also intended to stand for software, small, simple, and sufficient. The "S" also stands for "Practical", but the name "PPP" was already taken.

The approach described here has been used (in various variations and extremes) over a number of years at the University of Waterloo. SPP is an attempt to collect what we've learned in a form that can be used outside the university, and without any extraneous ornamentation.

2 Why Standards Aren't (Yet) Enough

In a perfect world, we would all have compilers that adhere to the appropriate standards, we would all have libraries that adhere to the appropriate standards, we would never want to do anything vendor-specific, wouldn't need to work-around hardware idiosyncrasies, and all vendor-provided software would be completely up to date, and, of course, completely bug free.

But, of course, this is not likely to be the case in very many non-trivial environments. There are many systems running with old software that may never be upgraded, OS vendors implement standards at varying rates of speed, and different bugs appear on different platforms. It is necessary to be able to work around incomplete or malfunctioning libraries and command sets, different versions of relevant standards, and vendors that are slow (or fast) in responding to change. SPP is intended to provide a way to work around these kinds of problems.

With luck, the need for SPP will lessen as time goes by, and it will get smaller and smaller, and eventually it will be obsolete. In the meantime, it can be a useful tool for software portability.

3 The Approach Used by SPP

SPP is a small collection of three types of objects:

1. C language include files
2. a small library of C language routines (`libspp.a`)
3. a small number of utility programs, typically implemented as Bourne shell scripts

The SPP include files provide a number of basic abilities. They allow a program to determine the architecture and version of a machine, in case the program needs to do something different on different machines. They allow indications of common differences between UNIX variants; for example the return type of `readdir()` is `struct direct *` on some machines and is `struct dirent *` on some other machines. They allow the definition of standard macros and types that may be missing on "older" machines. And they allow for the indication of certain machine attributes that are usually external to the C language interface, such as the name of the kernel, or the contents or location of various files (such as the `utmp` file).

For example, on a Sun machine running SunOS 4.1.3, `SPP_OS_SUNOS_4_1_3` is defined (via the include file `<spp/config.h>`). `SPP_OS_SUNOS_4_1_3` is used as the key to define other flag macros that indicate attributes of SunOS 4.1.3 (via `<spp/attrib.h>`), such as `SPP_HAS_UT_HOST`, which indicates that `struct utmp` has a `ut_host` field on this machine. (All macros defined by SPP include files start with `SPP_` to avoid potential conflicts with other software.) All the appropriate SPP include files are included as a result of including the file `<spp/spp.h>`.

The `spp` library provides a number of "common" routines that are not provided with the operating system on some machines. For example, the standard `strdup()` function is missing on some machines, so the `spp` library provides a version of it. The `spp` library on a given machine contains only those routines that are

needed to supplement that particular architecture, and so its contents vary from machine to machine, and it might even be empty on some very “standard” machines. The library isn’t intended to contain absolutely every routine that might be missing on a machine – it is intended to provide only relatively common, standard, routines that are likely to be useful in a variety of situations.⁴

The SPP utility programs are typically only those required to build and install software, and are usually simple cover scripts to cover variations across different architectures. For example, the *sppranlib* command shields the software author from having to know which machines have (or need) a *ranlib* command, or from having to write complicated tests in a *Makefile*.

Underlying the SPP implementation is the idea that things should be kept as small and as simple as possible, and that extraneous material should be kept to a minimum. With the basic *SPP_OS_** definitions and enough knowledge, a software author can write portable code. The other parts of SPP, such as the attribute definitions and the library routines, are merely convenience features to make it easier to write software that will work on machines that the author has no access to.

One major benefit of the SPP implementation is that a software author (or someone trying to port existing software) doesn’t have to use all of SPP; the programmer can use only those parts that are needed (or wanted) in a particular situation. For example, making existing software portable is sometimes only a matter of changing the original *Makefile* to use the *sppinstall* command to install the compiled program.

4 What SPP Doesn’t Do

SPP was created to help deal with a certain set of problems, and it isn’t intended to solve every portability problem that exists. For starters, it is very UNIX oriented. While a number of the concepts would port easily to other environments, it assumes an underlying tool set and filename syntax that is typically found only on UNIX systems.

SPP is oriented towards C language programming, and won’t solve all the portability problems that exist when writing in other languages, such as the UNIX shells, or *perl*. Some of the utility programs (such as *sppinstall*) can be useful in many situations, but the include files and the library routines are probably of use only to the C language programmer.

SPP doesn’t try to solve every portability problem, just the most common ones. For example, it doesn’t solve the problems inherent in trying to write software that will work on a variety of operating systems. But it does provide the basic building blocks that should allow a software author to make extensions that would help in solving less common problems.

SPP probably isn’t the portability solution for large, complicated systems. For example, SPP probably isn’t suited for use with the X Window System.

5 Makefiles, Install Scripts, and So On

SPP provides a small set of tools for use in building and installing software. Typically implemented as simple shell scripts, these tools are intended to augment the standard UNIX tools in simple but useful ways. The SPP tools are *sppmake*, *sppinstall*, and *sppranlib*.

sppmake is a simple wrapper for *make* that preprocesses an *SPPMakefile* and invokes *make* on the resultant *Makefile*. This is similar to *imake* (and was, in fact, inspired by it), but with two minor philosophical differences. In most instances, *imake* is used with a separate “template” file that often defines a large number of rules and macros for building the software. While this is useful in many cases, especially with large systems, it usually means that one can’t tell what *imake* will do by just looking at an *Imakefile*. *sppmake* is typically used to implement a small number of simple conditionals in an otherwise normal

⁴The *spp* library is similar to the GNU “liberty” library, but the latter includes additional, non-standard functions. The *spp* library is expected to shrink over time, and it seems likely that the liberty library will grow.

Makefile, e.g. to select different compiler options on different architectures. This means that, typically, *sppmake* is easier to start using than *imake*, since the *SPPMakefile* is more likely to resemble a normal Makefile. The other difference is that *imake* typically uses pre-defined *cpp* symbols in order to determine the current machine's architecture; *sppmake* typically uses an architecture-specific include file that defines appropriate SPP-specific symbols.

sppinstall is a "standard" install command, that allows the use of the same options and command name on multiple machines. For example, BSD-based machines have an *install* command, DEC OSF/1 machines have an *installbsd* command, IRIX machines have *bsdinst.sh*, and AIX machines have two different *install* commands, one in */usr/bin* and one in */usr/ucb*. *sppinstall* is just a version of the old, standard BSD *install* command that frees the software author from having to worry about what command name and arguments to use to get a program installed.

sppranlib serves a similar purpose. Some machines require the *ranlib* command to be used when creating libraries, some don't, and some don't even have a *ranlib* command. *sppranlib* looks for a *ranlib* command, runs it if it finds one, and does nothing otherwise. Again, this just makes it easier for a software author (or porter) to make things work on multiple machines.

These three simple tools seem to provide most of the functionality that is needed when building software on multiple, different machines. *sppmake* provides a simple way to customize the build process for different machines, and *sppranlib* and *sppinstall* provide useful utilities to make things easier for a software author.

6 Alternative Approaches to Portability

The problems that SPP attempts to address have been faced by many people and projects, and there have been many different approaches to solving (or avoiding) these problems. While SPP is not for everyone, it provides a simple approach with a low "start-up cost" i.e. it's easy to obtain and install, and it's simple to use. However, a comparison with alternate portability approaches may be useful.

One simple approach is for a software author to provide the necessary "missing" functionality. For example, since some machines do not include a *strdup()* function, an author could write a version of *strdup()* and include it with the software. An extension of this would be a custom support library for use in building the software. SPP essentially does this already, and the advantage of using SPP in this case is that the results can be shared by many projects, authors, and machines. SPP also frees an author from worrying about differences between machines, so that the author's effort can be directed to the program itself, rather than to the deficiencies that exist in some environments.

Another, more involved, approach is the "if your tools don't work, get different tools" approach. The portability problems that occur due to different libraries and commands can be solved by replacing the offending libraries and commands. For example, an author could adopt the GNU tool set, *gcc*, *make*, *install*, and the GNU C library, and many of the basic problems would be solved. An author would still need to deal with machine-specific differences (like the *utmp* file format and location), but the tools and library routines would be consistent. This approach doesn't address the problem of avoiding bugs on different systems, but with "identical" libraries, this is less likely to be a problem. This "replacement" approach (like SPP and like most others) requires that the replacement tool set be available anywhere an author wants to be able to build and install the software, and, if the software is to be distributed, this can place a major additional burden on the users of the software. SPP faces this same problem, but installing SPP is (intentionally) a smaller task than installing a complete set of replacement tools.

There are configuration tools, like GNU *autoconf* and Larry Wall's *metaconfig*, that provide a way to automatically create configuration scripts that will "research" a machine and generate an appropriate Makefile and other configuration information. These tools are very useful for larger projects (like *gcc* and *perl* for example), but they appear to be overkill for many smaller projects. There are many worthwhile and useful programs that require only a small amount of portability assistance, and for which the learning curve and configuration requirements of this kind of tool might be daunting for even an experienced software author. Another disadvantage to this kind of tool is that they can create Makefiles that are incomprehensible to

many people – it is usually valuable to be able to type *make -n install* and be able to understand what it is about to do to a machine.

One other approach to portability is that used in the X Window System source distribution; using *imake*, large template files, and the `<X11/Xos.h>` include file. `<X11/Xos.h>`, which tries to isolate many of the machine-specific differences in one common file, has much in common with parts of SPP, but it tends to be X-specific, rather than oriented towards general portability problems (and there is no guarantee that `<X11/Xos.h>` won't change in the next release). The X approach, in essence, created a large, self-contained, X-specific environment for building and installing programs, which is probably appropriate for a system of that size. But it did result in the build and configuration tools being hard to understand, and the average user or installer is unlikely to take the time to fully understand these tools. It also resulted in a system that is not trivial to adapt to other projects, and so *imake* has tended to be used only with X-related software.

The primary advantage of SPP is that it is simple, easy to learn and use, and that, by and large, it allows software authors to continue to use the standard UNIX tools in familiar ways, without having to make large-scale changes in order to deal with (sometimes) minor portability problems.

7 SPP Implementation

The SPP implementation has deliberately been kept as simple as possible, and can be installed using only the tools that should already be on every UNIX system.

The SPP include files are kept in a separate *spp* directory, in order to avoid name clashes with other files. There are currently 5 main SPP include files:

- `<spp/config.h>` is the file that causes the correct `SPP_OS_` macros to be defined for the current machine. This file is usually modified by each installation to include the appropriate sub-file. For example, on a machine running SunOS version 4.1.2, `<spp/config.h>` will include the file `<spp/config/sunos/412.h>`.
- `<spp/attrib.h>` uses the `SPP_OS_` macros to determine which “attribute” macros to define, e.g. `SPP_HAS_UT_HOST` is defined on most machines, but not on those that don't have a “host” field in the `utmp` file.
- `<spp/types.h>` provides additional standard typedefs, like `time_t`, if they are missing on particular machines.
- `<spp/lib spp.h>` declares the routines in the *spp* library, since if they are missing on a machine, then they are unlikely to be declared in the standard include files. The routines are declared only if they are included in the *spp* library on a particular machine, in order to avoid overruling or clashing with the real (correct) declarations.
- `<spp/spp.h>` is a cover file that includes all the others, and should normally be the file included in a program.

All names defined by SPP, with the exception of standard library routines provided as part of the *spp* library, start with `SPP_` to avoid name clashes with other software.

The `<spp/config.h>` file causes the macros `SPP_OS_MAJOR` and `SPP_OS_MINOR` (and, when appropriate, `SPP_OS_TEENY`) to be defined to indicate the version of the operating system in use, as well as specific macros naming the OS. For example, on a machine running SunOS version 4.1.2, the following would be defined:

```
#define SPP_OS_MAJOR    4
#define SPP_OS_MINOR    1
#define SPP_OS_TEENY    2
```

```
#define SPP_OS_SUNOS
#define SPP_OS_SUNOS_4
#define SPP_OS_SUNOS_4_1
#define SPP_OS_SUNOS_4_1_2
```

The OS version is defined in two different ways, making it easy, for example, to test for a particular version

```
#if defined(SPP_OS_SUNOS_4_1_2)
    /* do something special on SunOS 4.1.2 */
#endif
```

or to test for a version later than a particular version

```
#if defined(SPP_OS_SUNOS) && (SPP_OS_MAJOR >= 5)
    /* Oh, oh - things are very different now ... */
#endif
```

Note that `<spp/spp.h>` will likely cause C language declarations to be included, so it shouldn't be included by an SPPMakefile - `<spp/config.h>` should be used in an SPPMakefile.

The utility programs, *sppmake*, *sppinstall*, and *sppranlib*, are implemented as Bourne shell scripts, and are installed by simply copying them into the destination directory. They are deliberately kept as simple as possible. For example, the current version of *sppranlib* is

```
#!/bin/sh

if [ -f /usr/bin/ranlib ]; then
    exec /usr/bin/ranlib ${1+"$@"}
fi

exit 0
```

although the test might have to be extended to handle alternate locations for the *ranlib* command.

sppmake takes an SPPMakefile as input, runs it through *cpp*, puts the output into a temporary file called *#sppmaketmp* (for reference in case of problems when running *make*), and invokes *make* on that makefile.

The *spp* library is probably the most complicated part of SPP, because there is actual C code involved, and a slightly complex SPPMakefile that tries to include routines in the library only when that routine is missing from the standard libraries on a machine. Most of the routines in the *spp* library have been plundered from other freely-distributable libraries.

It is important for the SPP source to build and install as easily as possible, and for it to be almost completely self-contained, because if it gets too complicated, it will no longer be doing what it was intended to do.

8 Hints on Source Code Portability

Writing portable software isn't always an easy task. Even if we ignore different window systems and user interfaces, and non-UNIX operating systems, it still isn't an easy task. A software author needs to be aware of the applicable standards (like ANSI C for example), and has to be aware of potential portability pitfalls, usually before starting to write the software.

The C preprocessor and the `#if` and `#ifdef` conditionals are a major advantage when trying to write portable code. But, conversely, the misuse of `#if`'s and `#ifdef`'s can actually impede portability. For example, there is a common tendency to do things like

```

#if defined(sun) || defined(vax) || defined(sequent)
    /* blah blah blah */
#else
    /* some other blah blah blah */
#endif

```

and to have that kind of construct sprinkled throughout the code. The result is that porting to a new machine requires going through all of the code, searching for places to add yet another identifier to the appropriate `#if`'s.

Another common problem is the assumption that a `long` or an `int` is a particular size, or that bytes or bits appear in a particular order. Now that 64-bit processors (like DEC's Alpha chip) are becoming more common, there is starting to be a sizeable amount of code that looks something like this

```

#ifdef alpha
    int value;
#else
    long value;
#endif

```

And, historically, byte-ordering has often been determined by something like this

```

#ifdef vax
    /* little endian */
    ...
#else
    /* big endian */
    ...
#endif

```

which has proven to be a bad way of doing things.

The definitions in SPP's include files are, in part, an attempt to promote "healthy" coding styles. The attributes defined in `<spp/attrib.h>` encourage the use of conditional tests based on the attributes of a machine, rather than on the set of machines that happen to have that attribute. For example, this code

```

#if !defined(mips) && !defined(iris)
    strncpy( ut->ut_host, hostname, sizeof(ut->ut_host) );
#endif

```

is harder to maintain and port to new machines than this code

```

#ifdef SPP_HAS_UT_HOST
    strncpy( ut->ut_host, hostname, sizeof(ut->ut_host) );
#endif

```

especially if the `utmp` file entries are manipulated at multiple places in the code.

The problem of what type of variable to use is addressed in part by SPP's `<spp/types.h>` file. Historically, we've all known that the `time()` function returned a `long`, but using the newer, standard, `time_t` return value instead means that the code is more likely to run correctly on a 64-bit machine, other future machine types, or with future versions of `time()`. `<spp/types.h>` should ensure that all "standard" types are available to software authors on every machine, and, with luck, will serve as a gentle reminder to authors to define their own types, in their own header files.

Another important technique that leads to more portable software is the use of include files to isolate machine-specific tests and differences. If software has `#ifdef SPP_OS_SUNOS` in more than one file, or

in the middle of a bunch of code, it is probably an indication of a portability problem waiting to happen. If all machine dependencies and machine-specific `typedef`'s are isolated in one file (with a fairly obvious name, like `attrib.h` or `machines.h`), then porting to a new machine (or a new OS version) will usually require changing only that one file.

Outside of the program code itself, there are portability problems (and understandability problems too), often found hiding in a package's `Makefile`. The most common problem is the need to specify machine-specific options in the `Makefile` itself, which means it's impossible to compile source on multiple platforms without manual intervention. Another problem is `Makefile` commands that are in fact miniature shell scripts, usually in an attempt to cope with inconsistencies in OS vendors' command sets. *sppmake* provides a solution for the former problem, and small utilities like *sppranlib* and *sppinstall* provide ways to deal with the latter. The result is a `Makefile` that is far more understandable, more maintainable, and more portable.

9 Potential Extensions to SPP

It seems likely that SPP will be (should be) extended to provide additional attributes and library routines as it gets used in more situations. There may be a tendency to want to provide for every possible attribute or machine difference, but there is a danger that adding too much to SPP will obscure the "simple" part of SPP, and make it hard to use. Each addition to SPP will have to be weighed against this possibility.

It is possible that, as more people use SPP, it will become obvious that SPP is missing some important functionality, or that it "missed the boat" in some important area. Experience suggests that the relatively small set of tools that SPP provides will be sufficient for a large number of small to medium size programs, and so the expectation is that SPP will still be useful even if there are a number of situations where SPP isn't useful.

As SPP is extended, software packages may be built that require more recent versions of SPP. SPP defines `SPP_VERSION` and `SPP_RELEASE` to allow authors to check for particular versions of SPP in their software.

10 Conclusions

SPP provides a reasonable approach to portability, based on some relatively simple files and tools. It provides an easy way to write portable software, or to port existing software to new machines or environments.

SPP isn't necessarily (and probably isn't) the first place any of these portability problems have been addressed, and it probably doesn't introduce any radical new approaches. What it does attempt to do is to bring together a small collection of useful (and, hopefully, sufficient) tools that can solve many of the most common portability problems, in a way that can be adopted by many authors and installations.

However, SPP is not the right tool to solve all portability problems. For example, it is probably not suited for very large pieces of software, such as *gcc* or the X Window System distribution. It provides a tool set and basic approach for software authors; if SPP itself doesn't suit the needs of a particular project, it should provide useful examples and approaches to solving particular portability problems.

11 Acknowledgements and How to Obtain SPP

SPP is based on tools and ideas developed over the years at the University of Waterloo. SPP is a somewhat formalized, documented, and non-Waterloo-specific collection of the software and ideas. Accordingly, credit for the good ideas in SPP belongs to the UNIX software support staff at the University of Waterloo, and particularly the "software guys" in the Mathematics Faculty Computing Facility.

The SPP distribution is available, via anonymous FTP, from the machine `math.uwaterloo.ca` in the file `pub/spp/spp.shar.Z`. Contact the author (at `jmsellens@uwaterloo.ca`) if other arrangements are necessary.

The current version of SPP defines a relatively small, but common, set of machines. It is expected that this set will grow as other sites install SPP and provide appropriate information about other machines and architectures.

A Installing SPP

Installation of SPP has deliberately been kept simple, and uses only the lowest common denominator tools that should be on every UNIX machine.

The steps for installing SPP are:

1. Extract the source from the archive into an appropriate directory.
2. Change `spp/config.h` so that the appropriate machine and OS-specific configuration file is included on your system(s). If you have only one machine, simply `#include` the appropriate configuration file. If you have multiple machines of different types or versions, you may either have a separate `spp/config.h` file for each type of machine, or you may `#ifdef` as necessary to suit your installation and environment.
3. Change `Makefile` to set the destination locations of the SPP components. The `Makefile` suggests typical locations; you will want the commands to end up in your path, and the include files and libraries to end up where your compiler can find them, hopefully without extra `-I` or `-L` options.
4. Type the command `make install` and SPP should be ready to use.

If you have a system that is not currently covered by SPP, please contact the author to arrange for its inclusion. After an initial phase where many machines are added, SPP will, with a little luck, become usable on almost any UNIX system with very little customization needed.

B Examples of Use

Because SPP has been kept simple with independent components, some simple examples should be all that is necessary to demonstrate some typical uses of SPP.

An example of a simple `SPPMakefile` is:

```
#include <spp/config.h>

BINDIR = /usr/local/bin

CFLAGS = -O
#if defined(SPP_OS_AIX)
/**/# we want the BSD-ish version of things on AIX
CFLAGS = -O -D_BSD
#endif

all: prog

/**/# we use -lspp to pick up any missing "standard" routines
prog: prog.o
    $(CC) $(CFLAGS) -o prog prog.o -lspp

install: prog
    sppinstall -c -s -m 755 prog $(BINDIR)/prog
```

```
clean:
    rm -f *.o prog core
```

Note that the SPPMakefile includes `<spp/config.h>` to avoid getting the C language code that would be included by `<spp/spp.h>`.

An example of a program fragment that makes use of an SPP attribute is:

```
#include <stdio.h>
#include <utmp.h>
#include <time.h>
#include <spp/spp.h>

setutentry( ut )
struct utmp *ut;
{
    strncpy( ut->ut_line, "unknown", sizeof(ut->ut_line) );
    strncpy( ut->ut_name, "unknown", sizeof(ut->ut_name) );
#ifdef SPP_HAS_UT_HOST
    strncpy( ut->ut_host, "unknown", sizeof(ut->ut_host) );
#endif
    ut->ut_time = time( (time_t *)0 );
}
```

Note that this code includes `<spp/spp.h>` to get all the SPP-related include files, rather than just the configuration information required by the SPPMakefile.

References

- [1] Sellens, John, "Software Maintenance in a Campus Environment: The Xhier Approach", *LISA V Proceedings*, September 1991, pp 21-28.

CREATING A CONFIGURABLE COMPILER DRIVER FOR SYSTEM V RELEASE 4

*John F. Dooley
Vince Guarna*

Motorola Computer Group
1101 E. University Avenue
Urbana, Illinois 61801
jdooley@urbana.mcd.mot.com
vguarna@urbana.mcd.mot.com

ABSTRACT

This paper discusses the design and application of the configurable C compiler driver component of System V Release 4. The configurable compiler driver is a table-driven `cc` command that allows users to define include file, library, and tool paths as well as command line option translations and defaults.

The configurable driver presents several advantages over more traditional compiler drivers. The resulting compilation system allows many compilers with differing command line interfaces to be used in large system builds with no makefile changes. It allows the user to easily switch among several different C compilers, while presenting the user with a single command line interface. By allowing changes to compiler component paths, it makes the development of cross-compilers easier. Finally, it allows easy user configuration for the inclusion of default options.

We discuss the motivation for the project, the architectural design, and applications of the driver, including experiences with using the driver for System V Release 4 system builds.

INTRODUCTION

The configurable compiler driver (CCD) is a driver for C that can serve an unlimited number of compilation environments for System V Release 4 (SVR4). CCD uses an ASCII configuration file to look up individual "personalities" that specify various aspects of its behavior including

- (1) Component versions (preprocessor, compiler, optimizer, assembler, linker)
- (2) Include file and library paths
- (3) Command line option translations
- (4) Default options
- (5) Default `#defines` and `#asserts`

By setting a single environment variable, the user can determine which of the standard compilation configurations will be used when the `cc` command is used. Additionally, a second environment variable can be set to reference an arbitrary configuration file if the configurations supplied with the system do not meet the user's needs.

Requirements and Design Goals

The motivation for CCD emerges from several product requirements. First, Motorola's SVR4 is shipped with two C compilers — GNU (the default, chosen for its superior performance), and AT&T's C Issue 5 (CI5) compiler. Because we anticipated the need to use both compilers at various times, we needed a mechanism to switch between them conveniently.

Second, we anticipated the need to use cross compilation tools on 88K machines to reduce build times for 68K objects. This cross compilation environment effectively introduces another compiler to the language tool suite and further amplifies the need for a transparent switching mechanism.

Third was the need to develop a cross compilation environment for Motorola's real time products that was portable across both hosts and targets.

The most important requirement was complete compatibility with the existing System V Release 4 C compiler driver and its command line interface. A driver that required makefile changes would cause significant trouble for the SVR4 build team and would also be considered unacceptable by the user community. Therefore, the CI5 command line interface was chosen as the input language for the driver.

The fifth requirement was that the driver must exhibit the expected default behavior. Although the driver allows environment variables to be set to effect specific types of behavior from the compilation system, most users are not expected to use this functionality. Users familiar with C compilers are not likely to reference system documentation on the configurable system and will therefore not set the associated environment variables. The driver handles this gracefully by automatically referencing the default configuration file in this case.

Finally, CCD must be user customizable. Rather than build a simple merged driver that would serve the two SVR4 compilers with fixed, specific behavior, we decided that we would make runtime configurability available at the user level. Consequently, the driver's characteristics are completely determined by an ASCII file that can be modified by system administrators and overridden by individual users.

Implementation Decisions

The above requirements, along with a very limited time frame for implementation, led to the following implementation decisions. The driver would be table-driven, with a separate *translation table* for each compiler available on the system. The tables would be in a text configuration file that the user could copy and change. An environment variable would re-direct the driver to the new configuration file. A second environment variable would direct the driver to the correct table within the configuration file. There would be a simple translation language used with a few, simple keywords and control structures. The default table for translation would be the first table in the configuration file. For those compiler options that did not exactly match the CI5 command line interface, there would be a mechanism to pass them directly to the appropriate compilation phase.

ARCHITECTURE

Structure of the Driver

The configurable driver is broken up into four sections,

- (1) the main program,
- (2) the initialization section,
- (3) the translation section, and
- (4) the execution section.

Main

The main program is charged with driving the other three sections, with parsing the user-supplied command line, determining which compilation components to execute, and with controlling error reporting and cleaning up after compilation. The structure of the main program is shown in Figure 1.

Main:

```
Find the configuration file;
Initialize the translation table
and component strings;
Get the first user-supplied option;
While (there are still options to translate) do
    Switch on each option, translating the option into
    the equivalent option for the target compilation system.

    The translated option is placed on the argv list
    for the appropriate target compilation component.

    If the option does not begin with a minus sign,
    assume it is a file name
    and put it onto the input file list.

    If it is an unknown option,
    add it to the linkage editor's argv list.

    If an option that is supposed to take an argument
    doesn't have one, then report the error and continue.
end-while;

For (each of the files on the input file list) do
    Preprocess the file if necessary,
    Compile the file,
    Include profiling code if requested,
    Optimize the file if requested, and
    Assemble the file if requested.
end-for;

If no errors occurred above, and if the -c option was not used,
then link the file, clean up, and exit.
```

Figure 1. Configurable Driver Main Program

Initialization

During the initialization section, the configuration file is read in, the component names are defined, the component default strings are initialized, and the translation table is created. The configuration file contains one or more configuration tables, where each table contains all the information necessary to translate the configurable driver's option set into the equivalent options for the target compilation system.

No translations are done at this point, the translation table is just created, with each entry in the table being a sentence in the translation language. The set of indices into the translation table is the set of options recognized by the configurable driver. There may be options in the target compilation system that do not have translations from the configurable driver set. For example, the GNU -m machine dependent options have no equivalents in the driver option set, and are

thus not directly included anywhere in the GNU translation table.

Translation

In the translation section, the user-supplied command line option is translated into the target option string and the option string is returned to the main program. The translation sentence is extracted from the configuration table that was created in the initialization section. The parser is a traditional recursive-descent parser. The translation language that is parsed is a small structured language, with simple conditional and sequential control structures and 32 keywords. It's syntax is similar to C. The top level syntactic unit is the *sentence*, which is composed of three parts:

driver-option statement .

A more detailed description of the language is found below in the section on *Translation Expressions*.

Execution

When all user supplied options are translated, the driver goes through the list of input files and executes the components on each one, one at a time. It builds up each command line using the component name, translated pre-options, any silent or hidden options, the user's options and file, and the translated post-options. It then executes each component. Once all input files have been preprocessed, compiled, optimized, and assembled, the command line for the link editor is built and that component is executed.

CONFIGURATION FILE ORGANIZATION

The configuration file contains one or more configuration tables, each table including all the details necessary to translate the configurable driver options into the equivalent options for the target compilation system. The default SVR4 configuration file is located in */usr/ccs/lib/.compiler*, and contains tables for the GNU and C15 compilation systems.

Each configuration table has four parts,

- (1) the name of the table,
- (2) a section of component default specifications,
- (3) a section of component name declarations and pre- and post-options, and
- (4) a section of translation expressions.

Each of the last three table sections must be separated by a blank line.

Component Command Line Construction

Compilation command lines can be broken up into several distinct sections, but not all sections will be present for any given component invocation. The sections will always appear in the same relative order. The sections that appear depend largely on the user-supplied options, and occasionally on the values of user-supplied options.

For these reasons we have broken up the component command lines constructed by the configurable driver into several sections

- (1) the component's name,
- (2) a set of pre-options that are always added to the command line before any other options,
- (3) one or more sets of other conditionally included pre-options,
- (4) the user-supplied options and file names,

- (5) conditionally supplied post-options, and
- (6) post-options that are always included after all other options.

Component Default Specifications

The component default specifications are the conditionally included pre- and post-option sets placed on the command line. A default specification is a sentence in the translation language used by all objects in the configuration table. These default specifications invariably depend on which options the user has used on the `cc` command line. This means that these specifications are not translated until after all the user-supplied options are translated and we are constructing the component command line.

The results of these translations are generally called "hidden" or "silent" options because they are not usually visible to the user, and the user would be quite surprised to see them on the component command line.

Component Names and Pre- and Post-options

Component names are generally the absolute pathnames of each compilation component in the target compilation system. There are six possible components, preprocessor, compiler, profiler, optimizer, assembler, and linkage editor. Any of these may be omitted for a particular target; for example, the AT&T C15 compilation system has a merged preprocessor and compiler so in the C15 configuration table, the preprocessor component is omitted.

Component names can be changed by the user to use different pathnames. For example, if one is developing a new assembler (e.g. for a cross compilation environment) one can change the assembler component name and call the new assembler. This requires making a copy of the configuration file, making the necessary changes and using the copy. The user must then set the `CCMAP` and `CCCOMPILER` environment variables to point to the new configuration file and configuration table, respectively. This ability to easily change compilation systems while maintaining the same user interface is one of the major advantages of the configurable driver.

Pre- and post-options are those that are generally included unconditionally first and last on the component's command line. The pre- and post options are sentences in the translation language used by all objects in the configuration table. For any component either or both pre- and post-options may be omitted.

Translation Expressions

The translation language that is parsed is a small structured language, with simple conditional and sequential control structures and 32 keywords. Its syntax is similar to C. The top level syntactic unit is the *sentence*, which is composed of three parts:

driver-option statement .

A *statement* can be any one of the following:

- (1) a simple expression, which translates one configurable driver option into an equivalent target option. For example,

```
c %e %%.
V %e -v.
```

The first expression says to note when the `-c` option is used, but not put anything on the component command lines because `-c` is handled internally by the driver.

The second simple expression says to translate the `-V` driver option to the `-v` target option.

- (2) an *if-then-else* expression, which tests for the presence of a particular user-supplied option and includes one or more target options depending on the result of the test; since all

user-supplied options must have been translated before the *if-then-else* test can be made, *if-then-else* expressions are generally found in the component default specifications described above. For example,

```
%if (IG) %e "-lc".
```

says that if the -G driver option is not present, include -lc on the command line.

- (3) a multi-way *switch* expression, which tests the value of an argument to a user-supplied driver option and includes one or more target options based on the value of the argument. For example,

```
X %s (%a) {
    t: { %e -traditional }
    a,c: { %e -ansi }
    default: { %e %a } }.
```

This option says that if the user specifies the -Xt option, replace it with -traditional; if the user specifies either -Xa or -Xc, replace it with -ansi. If the command line option contains any other argument besides a, c, or t, it is ignored.

- (4) a *message*, which simply prints an informational message. An example of a message expression is

```
J %m "The -J option is not supported, skipping it".
```

- (5) a *block* of one or more statements, separated by semi-colons, and enclosed in curly braces.

Pre- or post-option specifications make good examples of block statements, as in this one for the GNU compiler

```
%dc /usr/ccs/lib/gcc-cc1
    "{ %e %t.i; %e "-quiet -dumpbase"; %e %f; }."
    ;
    "{ %if (#) %e -version;
        %if (S) %e -o %b.s %else %e -o %t.s }."
```

In this example %dc informs the parser that the following expression contains the name of the compiler. */usr/ccs/lib/gcc-cc1* is the name of the GNU compiler program. This is followed by a descriptor containing the pre-options for the compiler.

The descriptor is a block statement containing three simple expressions; the first includes the current temporary file name with a .i extension, the two GNU options *-quiet* and *-dumpbase*, and finally the name of the input file. The single semi-colon is the separator between the pre- and post-options.

The post-option descriptor is a block statement containing two *if-then-else* expressions. The first *if* replaces the *-#* driver option, if present, with the *-version* GNU option. The second *if* puts the string *"-o basename.s"* on the compiler command line if the *-S* driver option was used, and puts the string *"-o temporary-file-name.s"* on the command line otherwise. *BaseName* is the entire name of the current input file, without the extension. There is a translation expression for each of the configurable driver's 37 possible options.

The translation table for the GNU compiler is included in the Appendix.

PRACTICAL APPLICATIONS AND EXPERIENCES

Experiences with the configurable driver have shown it to be a valuable aid in compilation. One area is the build process for SVR4. Early baselines of this system were built with the C15 compiler. The change to GNU C as the default compiler was made early in the development of Motorola's version of SVR4. During that transition, no compiler-specific makefile changes were necessary to accommodate GNU C as called by CCD. This resulted in significant savings in build times and enabled the compiler team to focus on compiler problems, rather than build problems.

Interestingly, there were problems that arose from the use of GNU, but these problems were usually solved by adding new entries to the GNU translation table rather than changing makefiles. One example is GNU's handling of the `inline` symbol.

By default, GNU C recognizes `inline` as a keyword to specify function inlining. Additionally, this behavior can be turned off on the command line with the `-fno-inline` option. Treating `inline` as a keyword would normally have no effect except for the fact that a few source files in the SVR4 command suite (`vi`, for example) use the symbol `inline` as an identifier — GNU C generates parse errors for these files. This problem is easily solved by adding `-fno-inline` to the list of default options in the GNU translation table; however, there is a conflicting issue. For performance reasons, it is desirable to use the inlining feature, particularly for kernel builds. It is therefore sometimes important to have the compiler recognize `inline` when experimenting with kernel changes. The result is that a second GNU table was added to the configuration file — one that operates conventionally (no keyword) and one that can be used for performance work.

We have done some experimentation using the driver for cross compilations on 88K SVR4 machines (generating 68K SVR4 binaries). Libraries, include files, a cross compiler, a cross assembler, and a cross linker were loaded onto an 88K machine and a configuration table named `gnu68` was created. By going into the source base for some of the SVR4 commands, binaries for both 88K and 68K targets were successfully generated from the same source base, using the same makefiles, just by changing the `CCCOMPILE` variable. Although many aspects of the system build process make complete cross compilation difficult, we are investigating ways to use 88K machines in conjunction with the driver to reduce 68K build times.

The configurable driver also provides a convenient mechanism to do other useful tasks. For example, a programmer may wish to make sure that all compilations are performed with debugging or optimization turned on. This is trivially accomplished by adding the appropriate flag (`-g`, `-O`) to a local version of the configuration file. Conversely, someone may want to make sure that debugging or optimization is never turned on. This is easily accomplished by mapping the appropriate option to null in the translation table. For large, hierarchical applications with many makefiles, this could save a significant amount of time.

Performance Considerations

Several tests were run to compare the performance of the configurable driver with the GNU driver and the C15 driver. All tests were run on an 88K SVR4 system, with 16MB of main memory, 2 88100 processors and 8 88200 CMMU's. All times were measured using the elapsed (real) time using the Unix `/bin/time` function, and all compilations used the `-O` compiler option. The test using `make` also used the `-c` option.

In the first test, a C file with an empty `main()` function was compiled. In this test the compilation using the C15 compiler with the configurable driver was approximately 7% slower than just using the C15 driver. When the compilation was done with the configurable driver using the GNU compiler, the configurable driver was approximately 14% slower than GNU.

In a larger test, the 022.li SPEC benchmark files were compiled using the `make compile` target for the benchmark. This benchmark contains 22 C program files ranging in size from several hundred to over thirteen thousand bytes in length. In this longer test the compilation using the C15 compiler with the configurable driver was only 2% slower than just using the C15 driver.

When the compilation was done with the configurable driver using the GNU compiler, the configurable driver was only 3% slower than GNU. Thus, the configurable compiler driver does not add appreciably to compilation times.

The extra time required by CCD is attributable to the I/O necessary to read the configuration file and build the translation table. This work all occurs once, during initialization. Work is under way to improve the performance of this section.

SUMMARY and FUTURE WORK

CCD has shown to be a useful tool in the compilation environment and we expect to enhance it and expand its use in several ways.

One enhancement we have made is the recognition and expansion of environment variables in the configuration file. This is particularly useful in Unix baseline building. Part of the bootstrap process in the SVR4 baseline building procedure includes the definition of the environment variable \$ROOT. This variable is used to control where include files are retrieved; further, the variable is changed during the build process. Because CCD does not recognize environment variable syntax in the configuration file, two tables are required for the GNU compiler instead of one, one with \$ROOT as the first value and one without it.

We also want to expand the use of the driver to other languages such as C++ and Fortran. Because of the likelihood of our supporting multiple offerings for these languages, the benefits of extending the driver for these cases are probably worthwhile. Additionally, the availability of configurable drivers for all of our language tools increases the ability of third party software vendors to introduce language products for our platforms in a seamless manner. This should result in the long-term enhancement of our programming environment.

Finally, we are making the configurable driver available for new and existing SVR3 platforms. This is especially useful for the 68K environment where a transition is being made to a new C compiler.

A new version of the configurable driver is being developed to address problems introduced by using the driver for cross development for Motorola real-time products. Pushing the limits of the original requirements for the driver has forced us to re-examine the driver and change it.

This new version (CCD2) is a complete re-write of the configurable driver that eliminates many of the limitations of the original. CCD2 is still table driven, but is not restricted to using the C15 command line interface - a limitation that became acute when we wanted to develop cross compilation products for the real-time market. It also removes the six compilation phase restriction of CCD1 and gives the user greater flexibility in tailoring the command line interface. Finally, it enhances the translation language by allowing for the complete expansion of environment variables, and for the use of user-defined variables.

REFERENCES

- [1] Holub, Allen I., *Compiler Design in C*, Prentice-Hall, 1990.
- [2] Wirth, Niklaus, *Algorithms + Data Structures = Programs*, Chapter 5, Prentice-Hall, 1976.

APPENDIX

/*

This is the 88k version of the configuration file.
Copyright 1991, 1992, 1993 Motorola, Inc.

*/

/* The default "gnu" table is for the GNU-C 2 compiler. */

Env gnu {

%PREDEFINES "-D__GNUC__=2 -D__m88k__ -D__unix__

-D__OPEN_NAMESPACES__ -D__CLASSIFY_TYPE__ -D__CLASSIFY_TYPE__

```

-D__m88000__"
%PREASSERTS "-Acpu(m88k) -Asystem(unix) -Amachine(m88k)"
%PREPROC "{ %e "-lang-c -trigraphs"; %if (O) %e "-D__OPTIMIZE__ ";
    %if (v) %e "-Wall -Wtraditional -pedantic";
    %if (X) %s (%a) {
        a: { %e %% }
        c: { %e "-pedantic -D__STRICT_ANSI__ " }
        p: { %e "-pedantic -D__STRICT_ANSI__ -D_POSIX_SOURCE" }
        x: { %e "-pedantic -D__STRICT_ANSI__ -D_POSIX_SOURCE
            -D_XOPEN_SOURCE" }
        %default: { %e %% }
    }
}."
%COMPILE "{ %e "-funsigned-bitfields -fwritable-strings" }."
%STARTUP_PATH "/usr/ccs/lib"
%LOADR1 "{ %e "%STARTUP_PATH/crti.o";
    %if (X) %s (%a) {
        a: { %e "%STARTUP_PATH/values-Xa.o" }
        c,p,x: { %e "%STARTUP_PATH/values-Xc.o" }
        n: { %e %% }
        %default: { %e "%STARTUP_PATH/values-Xt.o" }
    }
    %else %e "%STARTUP_PATH/values-Xt.o" }."
%LOADR2 "{ %if (q) %s (%a) {
    g: { %e "/usr/ccs/lib/gmon.o" }
    l: { %e "-lprof -lelf -lm" }
    p: { %e %% }
};
%if (Y) { %e "-Y P, "; %e %r }
%else %if (p)
    %if (K) %s (%a) {
        minabi: { %e "-l /usr/lib/ld.so.1" ;
            %e "-Y P,/usr/ccs/lib/minabi/libp:
                /usr/ccs/lib/libp:/usr/lib/libp:
                /usr/ccs/lib/minabi:/usr/ccs/lib:/usr/lib" }
        %default: { %e "-Y P,/usr/ccs/lib/libp:/usr/lib/libp:
            /usr/ccs/lib:/usr/lib" } }
    %else %e "-Y P,/usr/ccs/lib/libp:/usr/lib/libp:
        /usr/ccs/lib:/usr/lib"
%else %if (q) %s (%a) {
    g,p: {
        %if (K) %s (%a) {
            minabi: { %e "-l /usr/lib/ld.so.1" ;
                %e "-Y P,/usr/ccs/lib/minabi/libp:
                    /usr/ccs/lib/libp:/usr/lib/libp:
                    /usr/ccs/lib/minabi:/usr/ccs/lib:
                    /usr/lib" }
            %default: { %e "-Y P,/usr/ccs/lib/libp:/usr/lib/libp:
                /usr/ccs/lib:/usr/lib" } }
        %else %e "-Y P,/usr/ccs/lib/libp:/usr/lib/libp:/usr/ccs/lib:/usr/lib"
    }
    l: { %if (K) %s (%a) {
        minabi: { %e "-l /usr/lib/ld.so.1" ;
            %e "-Y P,/usr/ccs/lib/minabi:
                /usr/ccs/lib:/usr/lib" }
    }
}

```

```

        %default: { %e %% } }
    %else %e "-Y P,/usr/ccs/lib:/usr/lib"
}
}
%else %if (K) %s (%a) {
    minabi: { %e "-l /usr/lib/ld.so.1" ;
        %e "-Y P,/usr/ccs/lib/minabi:
            /usr/ccs/lib:/usr/lib" }
    %default: { %e %% } }
    %else %e "-Y P,/usr/ccs/lib:/usr/lib" }."
%STARTUP "{ %if (p) %e "%STARTUP_PATH/mcrt1.o"
    %else %if (q) %s (%a) {
        g: { %e "%STARTUP_PATH/gcrt1.o" }
        l: { %e "%STARTUP_PATH/pcrt1.o" }
        p: { %e "%STARTUP_PATH/mcrt1.o" }
    }
    %else %if (IG) %e "%STARTUP_PATH/crt1.o" }."

%dp /usr/ccs/lib/gcc2/cpp "{ %e "-nostdinc";
    %e -undef; %if (# %or V) %e -v }." ;
"{ %if (X) %s (%a) {
    a,c,p,x: { %e %% }
    %default: { %e "-Dm88k -Dunix -Dm88000" }}
    %else %e "-Dm88k -Dunix -Dm88000";
    %e "-l/usr/ccs/lib/gcc2/include -l/usr/include";
    %if (P) %e -o %b.i
    %else %if (E) %e "-o -"
    %else %e %t.i }."
%dc /usr/ccs/lib/gcc2/cc1 "{ %e %t.i; %e "-quiet -dumpbase"; %e %f }." ;
    "{ %if (# %or V) %e "-version";
        %if (S) %e -o %b.s %else %e -o %t.s }."
%da /usr/ccs/bin/as "{ %e -o %b.o; %if (# %or V) %e -V }." ;
    "{ %if (S) %e %f %else %e %t.s }."
%dl /usr/ccs/bin/ld ; "{ %if (Q) %e "-Qy";
    %if (# %or V) %e -V;
    %if (IG) %e "-L/usr/ccs/lib/gcc2 -lgcc -lc";
    %e "%STARTUP_PATH/crtn.o"}."

A %s (%a) { -: { %e %% }
    default: { %e -A %all } }.
B %e -B %a.
C %e -C.
c %e %%.
D %e -D %all.
d %e -d %a.
e %e -e %a.
E %e -E.
f %e %%.
G %e -G.
g %e -g.
h %e -h %a.
H %e -H.
I %e -I %a.
J %m "The -J option is not supported, skipping it".
K %s (%a) {

```

```

    PIC,pic: { %e -fpic }
    fpe: { %m "Ignoring the -Kfpe option" }
    mau: { %m "Ignoring the -Kmau option" }
    minabi: { %e %% }
    %default: { %e %% } }.

L %e -L %a.
I %e -I %a.
O %e -O2.
o %e -o %a.
P %e %%.
p %e -p.
Q %e %%.
q %s (%a) { g: { %e -p }
             l: { %e "-mlprof -g -a" }
             p: { %e -p }
             %default: { %e %% } }.

S %e %%.
u %e -u %a.
U %e -U %a.
V %e %%.
v %e "-Wall -Wshadow -Wcast-align -Winline
      -Waggregate-return -Wwrite-strings
      -Wcast-qual -Wpointer-arith -Wstrict-prototypes
      -Wmissing-prototypes -Wnested-externs
      -Wtraditional -Wconversion -pedantic".

W %e %%.
X %s (%a) { t,n: { %e %% }
            a: { %e -ansi }
            c,p,x: { %e "-ansi -pedantic" }
            %default: { %e %% } }.

Y %s (%a) { S,L,F,U: { %e %% }
            l: { %e "-nostdinc -I/usr/include -l."; %e -l; %e %r }
            P: { %e -Y; %e %a } }.

z %e -z %a.
# %e %%.
}

```


Jam — Make(1) Redux

Christopher Seiwald
INGRES Corporation
Seiwald@Ingres.Com
March 11, 1994

Abstract

Despite the progress of UNIX, the basic mechanism by which developers build their programs — *make(1)* — has remained at its core unimproved since its inception. Most notably, the *make* language has seen few improvements. Jam is a *make* replacement that uses an extensible, expressive language for describing ways in which files relate. This new language simplifies the description of systems, both small and large, and renders extending Jam's functionality not only possible but easy.

Jam exists now and runs on many UNIX platforms, VMS, and NT. It is freely available in the comp.sources.unix archives. As proof of concept, it has been used to build a very large commercial product, generating in a single invocation 1,000 deliverable files from 12,000 source files.

1. Introduction

The UNIX *make(1)* program [Feldman, 1986], which automates the building of targets from their source files, is widely used. Together with its compatible successors (*dmake* [Vadura, 1990], GNU *make* [Stallman, 1991], NET2 *make* [BSD, 1991], SunOS *make* [Sun, 1989]) and mutations (*cake* [Somogyi, 1987], *cook* [Miller, 1993], *nmake* [Fowler, 1985], Plan 9 *mk* [Flandrena], *mms* on VMS, etc.), *make* enjoys world domination in its capacity.

As *make*'s author, Stuart Feldman, noted, *make* itself is not suited for describing huge programs. This is arguably because the *make* language has one useful statement: the expression of a direct dependency among files. This makes for a clumsy, bottom-up description of how to build a system — describing large systems this way is unmanageable. *make*'s successors try to overcome this difficulty with sundry tricks: dependencies with wild-carded names matched against directory contents; parsing command output as Makefile syntax; macro expansions with and without the help of the C preprocessor, etc.

Jam is an attempt to replace *make*'s rule system, with its bottom-up language and wayward implicit rules, with an expressive language that makes it possible to describe explicitly and cogently the compilation of programs. The current practice of building source simply because it matched wildcards is unreliable in the face of debris left around by a careless programmer. A robust product should be built explicitly, and to make this palatable, Jam makes it easy to be explicit.

A typical sample of Jam's language as seen by end-users will serve to anchor its description:

```
MAIN prog : prog.c ;
LIBS prog : libaux.a ;
LIBRARY libaux.a : compile.c gram.y scan.c ;
```

This example invokes three rules that instruct Jam to build an archive from three source files, to compile a fourth source file, and to link it against the archive. All three rules, as well as all other rules given as examples in this paper, are stock ones that come with Jam (see *Jambase(5)* [Seiwald, 1993]).

2. The Jam Language

As with *make*, the most important statement in the Jam language is the expression of a relationship among files. With *make*, the relationship is a direct dependency; with Jam, the relationship is user-defined. The expression of such a relationship is:

```
<RULE> <targets> [ : <sources> ] ;
```

This statement is referred to as a rule invocation, with the name of the rule leading the statement. Except for a handful of built-in rules, the definition of a rule is user-defined. The <sources> are optional. Each of the three lines in the example above are rule invocations.

Because rules invoke each other, the expression of a user-defined relationship can result in other user-defined relationships being made among the same or different files. In the case of the example given, the MAIN rule will invoke the rules:

```
CC prog.o : prog.c ;  
LINK prog : prog.o ;
```

These rules (presumably) handle the cases of compiling an object module from a C source file and linking that object module into an executable.

2.1. Rule Definition

A rule is defined in two parts: the Jam statements to execute when the rule is invoked (essentially a procedure call); and the actions (*sh*(1) commands) to execute in order to update the targets of the rule. A rule may have a procedure definition, actions, or both.

A rule's procedure definition is given with:

```
rule <RULE> { <statements> }
```

This statement causes <statements> to be interpreted by Jam whenever <RULE> is invoked. The <targets> and <sources> given at rule invocation are available as the special variables \$(<) and \$(>) in the <statements> defining the rule. <statements> may be any of the Jam statements listed in this document. Carrying on the CC example, a definition for the CC rule might be:

```
rule CC  
{  
    DEPENDS $(<) : $(>) ;  
}
```

This particular rule definition simply arranges for the targets to depend on the sources, using the built-in rule DEPENDS (described below).

A rule's updating actions are given with:

```
actions <RULE> { <string> }
```

This causes the *sh* script <string> to be associated with the <targets> in any invocation of <RULE>. Later, if Jam determines that the <targets> are out-of-date, it will pass <string> to *sh* for execution. Jam expands \$(<) and \$(>) in <string>, but \$(<) and \$(>) in this case refer to the <targets> and <sources> after they have been bound to real file path names (see Binding, below). Finishing out the CC example, a definition of CC actions would be:

```
actions CC  
{  
    cc -c $(CCFLAGS) -I$(HDRS) $(>)  
}
```

2.2. Rule Effects

Rule invocations have no outputs or return values and, instead, do their job through three distinct types of side-effects. The first is when a rule's procedure invokes built-in rules to modify the target dependency graph. These built-ins will be discussed shortly. The second is when a rule's procedure sets variables. The third is the association of the updating actions with the targets, which occurs whenever a rule with updating actions is invoked.

2.3. Built-in Rules

There are six built-in rules, five of which modify the target dependency graph. None of these rules have updating actions. The built-in rules are:

DEPENDS, INCLUDES, ECHO, TEMPORARY, NOTIME, NOCARE

DEPENDS and INCLUDES take <targets> and <sources>. ECHO takes only <targets>. TEMPORARY, NOTIME, and NOCARE take only <targets> and mark them with attributes to indicate special handling when descending the dependency graph.

DEPENDS

The basic builder of the dependency graph: it makes <sources> dependencies of <targets>, just like the simple *make* ":" dependency. If <sources> are newer than <targets> (using file update times for comparison), or if <sources> are being updated, then the updating actions of <targets> will be executed.

INCLUDES

A variation on DEPENDS: it makes <sources> dependencies of any targets of which <targets> are dependencies. This example makes both *foo.c* and *foo.h* dependencies of *foo.o*:

```
DEPENDS foo.o : foo.c ;
INCLUDES foo.c : foo.h ;
```

ECHO

Just echoes its targets to the standard output, as a means for communicating with the user. Jam knows no fatal error, so the message emitted by ECHO can only be advisory.

TEMPORARY

Allows for intermediate targets to be missing and not updated if the final target is up-to-date. If a target marked TEMPORARY is not present, then it simply inherits its parent's time-stamp. TEMPORARY can be used for any temporary target, such as the short-lived object module that is to be part of a library archive.

NOTIME

Indicates that the target is not really a file and therefore doesn't have a time-stamp. Any updating actions are only executed if the target's dependencies were updated, rather than on the basis of time-stamp comparisons. NOTIME is used for pseudo targets such as "all" or "install", which have dependencies but don't actually get built themselves.

NOCARE

Indicates that the target may both be non-existent and not have any updating actions. This loophole is used to make up for the sloppiness of the header-file scanning.

2.4. Jam Variables

Part of Jam's programmability lies in its treatment of variables. As with *make* and *sh*, Jam variables are lists of strings, with zero or more elements. But unique to Jam, the result of variable expansion is the product of the variable values and literal constants in the token being expanded. An example helps here:

```
$(X) -> a b c
t$(X) -> ta tb tc
$(X)$(X) -> aa ab ac ba bb bc ca cb cc
```

This approach makes quick work of many normal variable manipulations: prepending a path name to a list of file names, prepending “-l” to a list of library names, appending a “,v” to RCS file names, etc.

Jam has a modicum of variable editing options to replace components of a path name and to subselect members of a list. These options are discussed in usable detail in Jam’s manual page [Seiwald, 1993].

Unlike *make*, Jam does not defer expansion of variables. When a variable is referenced, even to assign a new variable, the value is expanded at that time.

Jam variables have two scopes: global and target-specific. Global variables behave much as one might expect, holding their value until reassigned. Target-specific variables take precedence over global variables when the specified target is being bound (see below) or updated. The distinction between global and target-specific variables is made when the variables are assigned. The syntax for setting the two types is, respectively:

```
<var> = <value> ;  
<var> on <targets> = <value> ;
```

Target-specific variables have several uses. A simple one is to permit different compiler flag settings for different source files. In this way, the actions of the CC rule may be used to compile any C source file, with various flags (HDRS, CCFLAGS) being adjusted per-target. Other uses of target-specific variables will be discussed shortly.

2.5. Flow-of-Control

In addition to statements for defining and invoking rules and setting variables, the Jam language contains statements for flow-of-control and file inclusion. The statements are:

```
if <condition> { <statements> } [ else { <statements> } ]  
  
for <var> in <value> { <statements> }  
  
switch <value> { case <value> : <statements> ; ... }  
  
include <file> ;
```

The “if” statement does the obvious; the <condition> is the usual mix of comparison and logical operators applied to variables.

The “for” statement iterates over the elements of <value>, assigning the (global) variable <var> to each element and executing the statement block.

The “switch” statement executes the statement block whose case <value> matches the switch’s <value>.

The “include” statement sources another file containing Jam statements.

Jam neither needs nor desires a macro preprocessor. Making rule definitions and file inclusions normal statements obviates a macro preprocessor for conditional compilation, as these statements may appear within Jam conditionals. Further, preprocessing would require the Jam language to play “dodge-em” with the preprocessor semantics.

3. Binding Files

Jam can find source and target files in distant directories, much like the functionality of `VPATH` in GNU *make* and *dmake*.

By default, a target is located at the actual path of the target, relative to the directory of Jam's invocation. If the special variable `$(LOCATE)` is set to a directory name, Jam locates the target in that directory (correctly concatenating the value of `$(LOCATE)` and the target's path name). If `$(LOCATE)` is unset but the special variable `$(SEARCH)` is set to a directory list, Jam searches along the directory list for the target file (again, correctly concatenating the path names).

Jam makes available the bound target names by using them when expanding `$(<)` and `$(>)` for updating actions. Thus, a target can be referred to by a short, unrooted name when invoking a rule to define a relationship, but any shell commands manipulating the target see a path name usable from the current directory.

`$(SEARCH)` provides `VPATH`-like functionality, allowing Jam to be invoked in directories other than where the source lives, while `$(LOCATE)` liberates Jam from the directory tree altogether. With it, Jam can run anywhere.

By setting `$(SEARCH)` and `$(LOCATE)` properly, Jam can handle a variety of build environments. For example, read-only source trees can be handled by pointing `$(SEARCH)` at a read-only source code directory while pointing `$(LOCATE)` to a working directory. As another example, "sparse" source trees can be handled by having `$(SEARCH)` contain two directories: first the developer's own directory, which contains only the files he is editing, and then his group's directory, which contains the master copy of all source. Most importantly, much of any build environment can be encoded in the settings of `$(SEARCH)` and `$(LOCATE)`, which leaves the file names used in rule invocations free from environment.

The power of `$(SEARCH)` and `$(LOCATE)` is realized when these variables are set per-target rather than just globally. Each individual target file can potentially be found along different search paths. In practice, related files will have the same search path, but Jam can efficiently accommodate the degenerate case of having these variables set per-target. In this way, Jam can build whole source trees, with source files scattered across directories.

4. Header-File Inclusion

Jam handles the incidental dependencies caused when source files include other source files. To find such dependencies, Jam scans source files for header-file inclusions, using a regular expression pattern match [Spencer, 1986]. The regular expression is given in the variable `$(HDRSCAN)`. The result of the scan is not interpreted directly by Jam; to arrange the necessary relationship, Jam calls a user-defined rule named in the variable `$(HRRULE)`, with the scanned file as `<targets>` and the found header-files as `<sources>`. Usually, the definition of `$(HRRULE)` includes a call to the built-in rule `INCLUDES`, which updates the dependency graph appropriately. An example `HDRSCAN` that works for C preprocessor includes is:

```
HDRSCAN = "^#[ \t]*include[ \t]*[<\""](.*)[>\""].*\" ;
```

The combination of `$(HDRSCAN)` and `$(HRRULE)`, when set per-target, enables Jam to handle just about any include-file syntax or semantics. Unfortunately, this mechanism doesn't understand conditional includes (`#include` within `#ifdef`), and can produce bogus dependencies that must be crudely pasted over with the application of the built-in `NOCARE` rule.

5. Time-Stamps

Like *make* et. al., Jam uses time-stamps to determine when targets are out-of-date. Another possible design, a more forward-looking one, would have Jam taking file update cues from an integrated source management system. This was deferred for two reasons: first, it would require picking a source management system with which to work (or attempting to engineer a generic interface to source management systems); second, it would preclude using Jam as a drop-in replacement for existing uses of *make*.

The code in Jam that checks dependencies is isolated enough to be altered to work with a source management system. Internally, Jam already distinguishes between updates due to newer dependents and updates due to updated dependents.

6. The Base Rule Set

A collection of rules providing *make*-like functionality is supplied with Jam. Called Jambase, the file provides a dozen-odd rules for compiling and linking C source code. Different versions of Jambase exist for UNIX, VMS, and NT, all providing the same rule set.

Figure 1 lists the rules defined in the current Jambase (described comprehensively in *Jambase(5)*).

The last act of Jambase is to include a file called Jamfile from the invoking user's current directory. Using the rules defined in Jambase, the user's Jamfile enumerates the source files and their relationship to the targets to be built.

The Jambase and Jamfile files share the same language; only their purposes distinguish them. It is possible to write a special-purpose replacement Jambase that is totally self-contained and needs no directory-specific Jamfile. It is also possible to use any Jam syntax — including conditionals, rule definitions, etc. — in a Jamfile.

7. The Example

Returning to our earlier example:

```
MAIN prog : prog.c ;
LIBS prog : libaux.a ;
LIBRARY libaux.a : compile.c gram.y scan.c ;
```

This example invokes three rules that instruct Jam to build an archive from three source files, to compile a fourth source file, and to link it against the archive. All these rules are defined by the Jambase file and do most of their work by invoking other rules defined in the Jambase.

MAIN calls LINK to set up the relationship between **prog** and **prog.o**, then calls OBJECT to set up the relationship between **prog.o** and **prog.c**. OBJECT calls a rule specific to the file suffix, in this case, CC for **.c**. Along the way, the various rules invoke the built-in DEPENDS rule to set up the dependency graph.

MAIN image :	source ;	link executable from compiled sources
LIBS image :	libraries ;	link libraries onto a MAIN
UNDEFINES image :	symbols ;	save undefs for linking
SETUID image :		mark an executable SETUID
LIBRARY lib :	source ;	archive library from compiled sources
OBJECT objname :	source ;	compile object from source
HDRRULE source :	headers ;	handle #includes
CC obj.o :	source.c ;	.c -> .o
LEX source.c :	source.l ;	.l -> .c
YACC source.c :	source.y ;	.y -> .c
YYACC source.y :	source.yy ;	.yy -> .y
BULK dir :	files ;	populate directory with many files
FILE dest :	source ;	copy file
SHELL exe :	source ;	install a shell executable
RMTEMPS target :	sources ;	remove temp sources after target made
INSTALLBIN sources :		install binaries
INSTALLLIB sources :		install files
INSTALLMAN source :		install man pages

Figure 1 — Rules supplied with Jam

```

MAIN prog : prog.c ;
  DEPENDS exe : prog ;
  LINK prog : prog.o ;
    DEPENDS prog : prog.o ;
  OBJECT prog.o : prog.c ;
    CC prog.o : prog.c ;
      DEPENDS prog.o : prog.c ;

LIBS prog : libaux.a ;
  DEPENDS prog : libaux.a ;
  NEEDLIBS on prog = libaux.a ;

LIBRARY libaux.a : compile.c gram.y scan.c ;
  DEPENDS libaux.a : libaux.a(compile.o) libaux.a(gram.o) libaux.a(scan.o) ;
  DEPENDS libaux.a(compile.o) : compile.o ;
  OBJECT compile.o : compile.c ;
    CC compile.o : compile.c ;
      DEPENDS compile.o : compile.c
  DEPENDS libaux.a(gram.o) : gram.o ;
  OBJECT gram.o : gram.y ;
    CC gram.o : gram.c ;
      DEPENDS gram.o : gram.c
  YACC gram.c : gram.y ;
    DEPENDS gram.c gram.h : gram.y ;
    INCLUDES gram.c : gram.h ;
  DEPENDS libaux.a(scan.o) : scan.o ;
  OBJECT scan.o : scan.c ;
    CC scan.o : scan.c ;
      DEPENDS scan.o : scan.c
  ARCHIVE libaux.a : compile.o gram.o scan.o ;
  TEMPORARY compile.o gram.o scan.o ;

```

Figure 2 — Rule execution for the example rule invocations

LIBS is a rule that arranges for **libaux.a** to become a dependency of **prog**, and it sets the target-specific variable **NEEDLIBS** to let the actions of **LINK** know that **libaux.a** should be included on the link command line. **LIBS** has no actions of its own.

LIBRARY is a rule that sets up the (somewhat complicated) dependencies between the library **libaux.a**, its members, and the temporary object modules that are to be its members. It calls **OBJECT** to set up the relationship between each of the temporary object modules and their source files. It also calls the **ARCHIVE** rule to handle the archiving of the temporary object modules into **libaux.a**.

A more complete list of rule invocations seen by Jam for this example is given in Figure 2.

Probably lost in this litany of rules are some important features: the **OBJECT** rule, when presented with the task of making a “.o” file from a “.y” file, called both the **CC** and **YACC** rules. Note that this is considerably easier and more deterministic than *make*’s approach of making a “.o” from whatever happens to be available. Also, note that the **YACC** rule took advantage of the **INCLUDES** built-in to make sure the dependencies on the generated file are accurately registered.

Actually, the rule definitions include a few more machinations to give special variables sensible defaults. For source code, **\$(SEARCH)** is set to **\$(SEARCH_SOURCE)**; for object files, **\$(LOCATE)** is set to **\$(LOCATE_OBJECT)**; for C source files, **\$(HDRSCAN)** is set to the example pattern mentioned above, and **\$(HRRULE)** is set to “**HRRULE**”, the generic header-handling rule defined in the **Jambase** file.

8. Implementation

The weight of Jam's implementation is evenly divided between its rule-processing subsystem (driven by a *yacc*(1) grammar), its recursive binding and scanning subsystem, and its recursive build subsystem.

The rule-processing subsystem is entirely system independent, only setting in-memory variables, building the dependency graph, and associating update actions with targets. The *yacc* grammar is less than 200 lines.

The recursive binding and scanning subsystem is mostly system independent, but calls system-dependent routines to time-stamp files and to manipulate file names.

The recursive build subsystem is mostly system independent, but calls system-specific routines to execute shell commands (which are system-specific as well).

The system dependencies are hidden through three interfaces: one to time-stamp files; one to manipulate file names; and one to execute shell commands.

The file time-stamp interface has two layers: a higher one that asks about individual files; and a lower one that scans directories and library archives whole. The latter is more efficient, and all current implementations (UNIX, VMS, NT) are coded against it.

The file name manipulation interface consists of two routines: one to break a file name down into its components and one to build a file name from its components. These are quite simple — except on VMS, where concatenating path names is black art.

The shell-command interface currently approximates the UNIX *system*(3) call interface, with an addition for catching interrupts.

Jam achieves its functionality while going sparingly on features. It has only four flags (mostly to do with debugging), six built-in rules (DEPENDS, INCLUDES, ECHO, NOTIME, NOCARE, TEMPORARY) and six special variables (*\$(>)*, *\$(<)*, *\$ (SEARCH)*, *\$ (LOCATE)*, *\$ (HDRSCAN)*, *\$ (HRRULE)*). The whole of Jam for UNIX is under 5,000 lines of code, exclusive of Henry Spencer's *regexp*(3) regular-expression code (about another 1,300 lines).

A design goal of Jam was portability, specifically so that the same mechanism could be used to build the same system on different platforms. Jam scores well in this category: the OS interface is constricted, leaving the bulk of the system dependencies in the Jambase file. Even the Jambase file is somewhat portable, with only the filename syntax and the actual update commands having to change between UNIX and VMS. Jamfile files themselves usually contain nothing system-specific.

9. Performance

Used to build from scratch a large commercial software system (the INGRES relational DBMS), lapse time for Jam breaks down as follows (on an HP9000/710):

parsing 5,000 lines of Jamfiles	16 seconds
stat()'ing 12,000 source files	1 minute
scanning 12,000 source files for headers	9 minutes
actual building (compiling, linking, etc)	12 hours

The simple conclusion is that Jam's performance is inconsequential. When everything is up-to-date, only few improvements could be made. *stat()*'ing files is essentially unavoidable without resorting to other techniques for determining outdated targets. Scanning source files could be avoided by caching header-file dependency information in state files. SunOS *make* and *nmake* use this approach. The only other recourse is to hammer on the *regexp* implementation.

The real performance limitation is in actual building time. Jam does not yet support parallel command execution, which on a large SMP system can reduce build time by a factor of 5 or more. This feature is anticipated.

10. Comparisons

Jam's per-target variables are a convenience approached only by SunOS *make*'s "target := macro = value" syntax. Both Jam and SunOS *make* make use of the value when updating the target, but Jam gets added mileage out of the facility by using the value during the binding and header-file scanning.

Jam's searching mechanism is superior to VPATH in two ways: first, it provides not only searching for existing targets, but also binding for new targets; second, Jam's SEARCH and LOCATE variables can be set per-target. GNU *make* allows VPATH to be set selectively, using patterns, and the patterns could be full file names, but GNU *make* handles the degenerate case of separate values per file poorly. Jam's SEARCH and LOCATE mechanism can make the invoker's directory irrelevant, which amounts to a complete solution.

Jam's pattern-scanning method of header-file scanning is faster than those that offload the problem to separate programs (*dmake*, *cake*, GNU *make*). It is not strictly correct, like SunOS *make* and GNU *make*, which use the C preprocessor. Jam's mechanism, driven by per-target variables and user-defined relationships is, however, quite flexible. It can handle languages that don't offer a separate preprocessor, as well as languages where the result of a file being included is more than just a simple dependency. For example, when a *yacc* file includes a C header-file, Jam can be made to understand that the generated C source file will include the generated C header-file. Jam supports these types of arrangements entirely in its language.

Jam is missing a few features cherished by some *make* users: the ability to run update commands concurrently and fancy variable editing. These may appear in future versions of Jam.

11. Discussion

The comparison of Jam's language with *make*'s is somewhat subjective and complicated. As stated in the introduction, Jam is an attempt to replace *make*'s rule system with an expressive language that makes it possible to describe explicitly and cogently the compilation of programs.

In this respect, Jam is a success. For small systems, the Jamfile file is often not larger than the three lines that made up our example. For large systems, any added complexity can be centralized in the Jambase file, while the Jamfile file(s) in source directories remain simple.

Jam's rule semantics, that of expressing named relationships among files, is Jam's single biggest advantage over its contemporaries. Its power and economy of expression seem unmatched. There are two other approaches deserving mention. *nmake* and *dmake* allow new operators (replacements for the simple ":" dependency statement) to be defined as macros, and these can be used to create new relationship types. Unfortunately, the number of available "operator" characters is limited, and the coding of the macros would curl the eyebrows of even seasoned *sendmail* hackers. *cook*, *cake*, Plan 9 *mk*, and NET2 *make* promote a different approach: that of defining variables and then #include'ing "recipes" (other *make* files) that define the relationships. The recipes are the approximate equivalent of Jam rules, using pass-by-name variables. This scheme works, but it is an ugly ordeal to try to recover with a preprocessor the functionality that is lacking in a language.

The Jam language turns out to be fairly straightforward to program. With its reliance on keywords rather than special characters and its use of a ";" to terminate statements, it is easier reading than most *make* syntax.

Abandoning *make* syntax was an easy decision: even those new *makes* that understand traditional *make* syntax get their added functionality through incompatible syntax. If compatibility with *make* is the priority, users can just use *make*. If users want greater functionality, they can't use vanilla *make* anyway.

The proof of Jam is in the pudding (sorry...): it is worth mentioning that the timing information given above is for a single, non-recursive invocation of Jam to compile 12,000 source files scattered throughout 300 directories, producing 7,000 intermediate targets and 1,000 deliverable files. Each source directory contains a single Jamfile with an average of 1.5 words per source file (including the source file name). The author knows of no other *make* that can approach such completeness with such economy.

12. Availability

Jam is freely available in Volume 27 of the comp.sources.unix archives. It is known to compile and work on VMS (Alpha and VAX) and the following variants of UNIX: BSD/386, OSF/1, DGUX 5.4, HP-UX 9.0, AIX, IRIX 5.0, PTX V2.1.0, SunOS 4, Solaris 2, Ultrix 4.2, and Linux. Support for NT is not part of that initial release but is available from the author.

13. Bibliography

[Brokken, 1994]

Frank B. Brokken and Karel Kubat, "ICMAKE — the Intelligent C-like MAKer, or the ICce MAKE utility", Linux Sources, 1994

[BSD, 1991]

BSD NET2 make(1) manual page, BSD NET2 documentation, July 1991.

[Feldman, 1986]

S. I. Feldman, "Make — A Program for Maintaining Computer Programs", BSD NET2 documentation, April 1986 (revision).

[Flandrena]

R. Flandrena, "Plan 9 Mkfiles", available via anonymous FTP from plan9.att.com.

[Fowler, 1985]

Glenn Fowler, "The Fourth Generation Make", Proceedings of the USENIX Summer Conference, June 1985.

[Miller, 1993]

Peter Miller, "Cook — A File Construction Tool", Volume 26, comp.sources.unix archives, 1993.

[Seiwald, 1993]

Christopher Seiwald, Jam(1) and Jambase(5) manual pages, Volume 27, comp.sources.unix archives, 1993.

[Spencer, 1986]

Henry Spencer, Regexp code and comment, comp.sources.unix archives, 1986.

[Stallman, 1991]

Richard M. Stallman and Roland McGrath, "GNU Make — A Program for Directed Recompilation", Free Software Foundation, 1991

[Somogyi, 1987]

Zoltan Somogyi, "Cake, a Fifth Generation Version of Make", Australian Unix System User Group Newsletter, April 1987.

[Sun, 1989]

Sun Microsystems Corporation, SunOS make(1) manual page, SunOS 4.1.2 documentation, September 1989.

[Vadura, 1990]

Dennis Vadura, dmake(1) manual page, Volume 27, comp.sources.misc archives, 1990.

Writing, Supporting, and Evaluating Tripwire: A Publically Available Security Tool

Gene H. Kim and Eugene H. Spafford*

*COAST Laboratory
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398*

ABSTRACT

Tripwire is an integrity checking program written for the UNIX environment that gives system administrators the ability to monitor file systems for added, deleted, and modified files. First released in November of 1992, Tripwire has undergone several updates and is in current use at thousands of machines worldwide.

This paper begins with a brief overview of what Tripwire does and how it works. We discuss how certain implementation decisions affected the course of Tripwire development. We also present other applications that have been found for Tripwire. These unanticipated uses guided the demands of some users, and we describe how we addressed some of these demands without compromising the ability of Tripwire to serve as a useful security tool.

We also discuss the process of releasing, and then supporting, a widely available and widely used tool across the Internet, and how meeting users' high expectations affects this process. How these issues affected Tripwire, done as an independent study by an undergraduate, is also discussed. Software tools that were used in developing and maintaining Tripwire are presented. Finally, we discuss problems that remain unresolved and some possible solutions.

1 Introduction

1.1 Genesis

Tripwire is an integrity checking program written for the UNIX environment that gives system administrators the ability to monitor file systems for added, deleted, and modified files. Tripwire was primarily intended to be used for intrusion detection, and its design and operation are described in detail in [9].

The first version of Tripwire was completed in September 1992. Since then, its design and code have been available for use by the community at large. An intensive beta test period resulted in Tripwire being ported to over two dozen variants of UNIX, including several systems neither author

*Gene Kim is currently at the University of Arizona.

has yet to encounter. Currently entering its seventh (and possibly last) revision, we believe there are interesting lessons to be learned from our experience of design, distribution and support of Tripwire.

Tripwire was conceived somewhat accidentally in 1991. In the fall of 1991, Gene Kim was an undergraduate student at Purdue University looking for a research project. He approached Gene Spafford, who had been his professor in several CS courses, to ask about doing a research project. Spafford suggested that Kim do a project involving message-digest and cryptographic checksum algorithms for integrity management; hours prior to the visit, Spaf had been considering the problem of how to tell which files had been altered after a system break-in. Gene's visit to Gene¹ was thus fortuitous.

During the spring of 1992, Gene performed initial experiments with several digest algorithms to better understand how they worked. One of these experiments involved collecting the signatures² of tens of thousands of files on several machines at the university computing center. A small mistake in coding led to every existing file on the main campus instruction computers being touched, thereby being marked for backup, leading to a shortfall in backup media. This resulted in several panicked phone calls at 2 AM by computing center staff and some administrative repercussions — all much to Gene's chagrin and to Gene's amusement. Experimentation was suspended temporarily.

During the summer of 1992, Gene developed the initial Tripwire program during a summer internship. Upon his return to Purdue in the fall, and consultation with Gene, the program was revised, documented, tested, and distributed to beta testers. Officially released on November 2, 1992, Tripwire is now being actively used at thousands of sites around the world. Published in volume 26 of `comp.sources.unix` and archived at numerous FTP sites around the world, Tripwire is widely available and widely distributed.

1.2 Design

Ultimately, the goal of Tripwire is to detect and notify system administrators of changed, added, and deleted files in some meaningful and useful manner. These reports can then be used for the purposes of intrusion detection and recovery. Changed files are detected by comparing the file's inode information against values stored in a previously generated baseline database. Detecting altered files beyond inode attribute checking is provided by also storing several signatures of the file — hash or checksum values calculated in such a way that it is computationally infeasible to invert them.

A high level model of Tripwire operation is shown in Figure 1. This illustrates how the Tripwire program uses two inputs: a *configuration* describing the file system objects to monitor, and a *database* of previously generated signatures putatively matching the configuration.

In its simplest form, the configuration file contains a list of files and directories to be monitored, along with an associated selection mask (i.e., a list of attributes that can be safely ignored if changed). The database file is initially generated by Tripwire, containing a list of entries with filenames, inode attribute values, signature information, selection masks, and the configuration file entry that generated it.

Tripwire uses these files in all its modes of operation. These modes handle the building of the baseline database, the reporting of all changed, added, and deleted files by comparing a newly generated database against the baseline database, and the updating of the baseline database by replacing out-of-date entries.

More detailed descriptions of operation and features of Tripwire are given in [9] and [10].

¹For the remainder of this paper, both authors are referred to as "Gene" for purposes of symmetry and to more evenly distribute references of blame, chagrin, and scholarship.

²Throughout the Tripwire documentation, we refer to secure hashes, message digests, and cryptographic checksums under the generic term "signature."

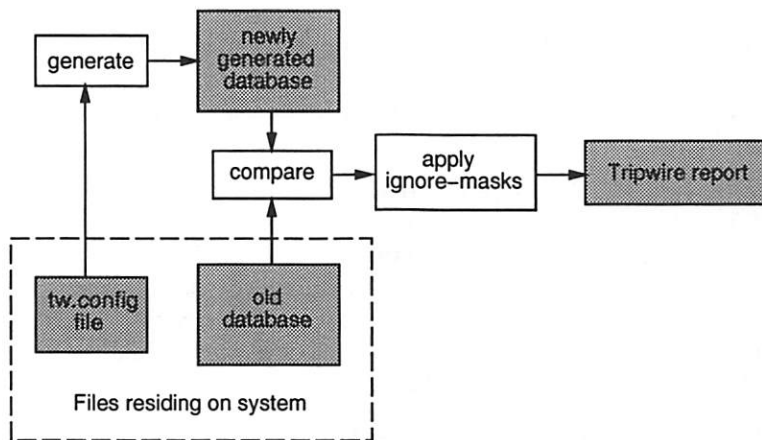


Figure 1: Diagram of high level operation model of Tripwire

2 Theory vs. Practice

As with most research projects, what we thought we wanted and what we finished with were somewhat different.

2.1 What we thought we wanted

Tripwire was envisioned by one of us (Spafford) as a tool to help with intrusion detection and recovery. It was to be a small tool (or set of tools) with a small set of functions. The initial scope of the project was to investigate design issues and to implement a prototype of the tool.

Based on previous experience with various tools, and after some discussion, we agreed that the following goals would be sought after in the design and implementation of Tripwire:

Functionality We wanted to build a tool that could be used to find unauthorized changes in a UNIX file system. The tool should be simple enough to use that most system administrators would (and could) use it.

Portability Among the primary requirements of Tripwire was that it was to be be publically available and freely redistributable. This implied that even the "lowest common denominator" of UNIX systems could build and run Tripwire, underscoring the perceived necessity of a highly portable design. This also meant that the database files for Tripwire would be readable text rather than some fixed-format binary records.

Configurability We wanted a design that would allow site administrators to select what files to monitor and what attributes to monitor. We knew that the security policies and needs of sites would vary considerably, and wanted Tripwire to support those differences.

Flexibility We wanted to make it possible to choose which signatures to generate, or to substitute new signature routines not part of the release. In particular, we want to make it possible for someone to add in one or more cryptographic checksum methods requiring a password for each execution.

Safety We wanted to build a system that required no special privileges to run, and that consisted of source that would be easy to read and understand. Spafford's previous experience with setting this as a design goal for the COPS system[6] has been well-received; system admins are more comfortable running security tools if they can examine the source and customize it if they feel the need.

2.2 What we actually got

Tripwire coding started near the end of the Spring 1992 semester. Semester deadlines for grades undoubtedly motivated some of the early implementation shortcuts that were made at that time. Implementation continued throughout the summer. An intensive testing period began in September 1992, involving two hundred users around the world who has responded to a USENET post.

The first official version of Tripwire was released in November 1992 on the anniversary of the Internet Worm. Since then, seven subsequent versions have been released to incorporate bug fixes, support additional platforms, and add new features. We estimate that Tripwire is being actively used at several thousand sites around the world. Retrievals of the Tripwire distribution from our FTP server initially exceeded 300 per week. Currently, seven months after the last official patch release, we see an average of 25 fetches per week. This does not include the copies being obtained from the many FTP mirror sites around the Internet.

We have received considerable feedback on Tripwire design, implementation, and use. We believe that version 1.1 of Tripwire has succeeded in meeting most of our goals:

Functionality Tripwire appears to meet of the needs of system administrators for an integrity checking tool. We have gathered reports of at least seven cases where Tripwire has alerted system administrators to intruders tampering with their systems. (Experiences with Tripwire for intrusion detection is presented in [10].) The continuing interest in new releases, and the endorsement of Tripwire by various response teams has also confirmed the utility of Tripwire.

Portability Tripwire has proven to be highly portable, successfully running on over 30 UNIX platforms. Among them are Sun, SGI, HP, Sequent, Pyramids, Crays, NeXTs, Apple Macintosh, and even Xenix. Although this has necessitated some awful hacks (some of which are detailed below), the user needs to do very little to configure for a specific machine.

Configurability Tripwire is being used at large homogeneous sites consisting of thousands of workstations, as well as at sites consisting of a single machine. Tripwire allows considerable flexibility in the specification of files and directories to be monitored. Specifying which file attributes can change without being reported allows Tripwire to run silently until a noteworthy filesystem change is detected.

Flexibility Tripwire includes seven signature routines to supplement the file inode information stored in the database to augment the ability to detect changed files in a non-spoofable manner. Because signature routines are often slow cryptographic functions, Tripwire allows system administrators to specify which signatures routines to use for files, and when they should be checked.

Users report that the ability to choose which signatures to use is appreciated and used. We have yet to receive a report of someone (other than ourselves) integrating a cryptographic checksum or a new message digest algorithm, however.

Safety Because Tripwire sources are publically available and freely distributable, they are available for scrutiny by the community at large. Possible weaknesses have been discussed in the literature (e.g., [15]) and by private communication (e.g., [2]). These evaluations were written when our design document was not yet publically available, and reflects positively that our sources are adequately readable.³

Furthermore, we have received many reports of system administrators modifying Tripwire, sometimes extensively, to suit their local site needs. In general, such changes have not necessitated changes throughout the sources. Instead, changes have been restricted to one or two files.

³All reported weaknesses have been addressed by changes to the code or documentation.

As time went on, we discovered that there was one important goal we had completely overlooked in our design: scalability. We failed to recognize the immense size of some installations, and the resulting problem of managing Tripwire data across hundreds of platforms.

Based on our experience with Tripwire, we would encourage designers of security tools to consider carefully all of these goals for their own efforts. We found that reference to these high-level goals helped us resolve questions when they arose, especially when we were evaluating some new feature or extension to be added to the code.

3 What we learned along the way

All my ideas are good, it's only the people who put them into practice that aren't.

Amos Brearly, character in British TV show[5]

In the previous section, we presented several key design aspects of Tripwire that have supported its widespread use. In this section, we present some of the key (mis)decisions of Tripwire development that significantly affected its development.

3.1 Providing portability

Tripwire was designed and written to run on any reasonably implemented UNIX operating system. The underlying philosophy was to write Tripwire for the lowest common denominator of all existing UNIX systems: building and compiling Tripwire should require only tools usually bundled with the operating system (e.g., K&R C compiler, `lex`, `yacc`). Furthermore, the Tripwire program was designed to be completely self-contained, using no system utility programs such as `grep` or `awk`. This restriction allows system administrators to run Tripwire on machines where the integrity of the system utilities may be in question (e.g., on machines with evidence of intruder tampering).

3.1.1 Theory

Several “meta-configuration” utilities exist, such as the `metaconfig` package originally written by Larry Wall. Generally these packages assist software writers by providing a utility that discovers any quirks or non-standard implementations of the underlying operating system and libraries. Using such a package, the programmer writes for one canonical interface, using the information gathered by the meta-configuration script to customize the code at compile-time.

Against the repeated advice of Gene, the other Gene decided to forego using `metaconfig`. His perception of the simplicity of the Tripwire sources coupled with the perceived complexity of `metaconfig` led him to believe that it would be less work to make small changes to the source code. Thus, provisions were made by grouping UNIX systems into two categories: those derived from AT&T System V and those derived from BSD. While this delineation may seem simple, this partitioning soon proved insufficient to allow straightforward compilation on most machines.

3.1.2 Implementation

In September 1992, we sent the first beta version of Tripwire to over two hundred testers. These testers tried Tripwire on over twenty different platforms, including Suns, HPs, IBMs, NeXTs, Sequents, Crays and PCs running Xenix. They then sent back those changes necessary to allow correct Tripwire compilation and operation on their respective platforms. Although most of the changes were easily merged back into our source tree, virtually all changes to the header sections (i.e., containing the `#include` directives) conflicted with each other.

Creating a framework for correctly including header files in the `/usr/include` hierarchy thus proved surprisingly tedious. Although these header files are intended to hide system-specific data structures from user programs, certain files are needed by virtually every program (e.g., `stdio.h`). However, names of many header files are not consistent across all platforms (e.g., `string.h` and `strings.h`), while others are absent on many machines (e.g., `stdlib.h`). Furthermore, on certain

machines, inclusion of two include files may be mutually exclusive. Because many machines used by the initial Tripwire testers (and the Unix-using population at large) predated standards dictating what data structures and definitions reside in which file (e.g., POSIX 1003.2), no assumptions about these include files could be made.

This fact led to the header sections being replaced by hand-generated `#ifdef` blocks, which have since been modified almost beyond recognition. The first test release of Tripwire allowed its compilation on BSD and System V implementations as interpreted by SunOS 4.1.1 and Solaris 1.0. As testers modified the header sections to allow its compilation on their platforms, its complexity grew rapidly. As a consequence of their changes to the header structure, another platform that previously had no problems compiling would often fail.

These *ad hoc* modifications of the header sections and repeated distribution and testing of merged changes eventually produced a reasonable framework. However, only the patience and persistence of our testers allowed this iterative method to succeed. The first several patches striving to add “portability” to the test version of Tripwire invariably would prevent correct compilation on many machines.

A header section extracted from a Tripwire source file is shown in figure 2.

3.1.3 Lessons learned

Tripwire now compiles “out of the box” on most platforms, and adding configurations for new platforms within this framework is relatively easy. However, the tedium required to build this framework is hard to justify when an existing tool could have automated this process.

The configuration framework in Tripwire remains versatile enough to allow the addition of system specific capabilities. Special file operations for Apollo Domain/OS and HP/UX CDF (Context Dependent Filesystem) have been added to Tripwire by users, and are now included in the Tripwire distribution without affecting users of older versions.

One clear choice we could have made was to have declared certain implementations as “unsupported” and thus ignored the portability questions. In one sense, this was done: to support some of the architectures on which Tripwire runs, we depend on our users to provide the necessary configuration information and specialized code. We have no access to the architectures involved to do the work ourselves.

3.2 You administer how many systems?

When Tripwire was distributed to beta testers in the September 1992, it did not have a built-in preprocessor. Because Tripwire was originally intended to monitor small numbers of files for changes, we did not envision any need for such a mechanism.

During the testing period and early deployment, numerous system administrators ran Tripwire on many machines at their site. Given large enough numbers of machines, the time required to configure Tripwire on all machines proved prohibitive. Exacerbating this is that their Tripwire configurations typically cover thousands of files, and that machine disk configurations are rarely uniform enough to support direct reuse.

Thus, a preprocessing language to allow Tripwire to share common configuration files among multiple machines was added. The impetus for this was when a system administrator wrote about his ambitions for running Tripwire on the 700 Sun workstations at his site — a monumental task without sharing, requiring generation of 700 different configuration files! We had not envisioned an environment like this when we originally designed Tripwire.

3.2.1 Implementation

To allow scalable use of Tripwire at large sites (e.g., up to thousands of heterogeneous machines), support was added to allow the reuse of configuration files. This was implemented by adding

```

#include "../include/config.h"
#include <stdio.h>
#ifdef STDLIBH
#include <stdlib.h>
#include <unistd.h>
#endif
#include <fcntl.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/stat.h>
#ifndef NOGETTIMEOFDAY
# include <sys/time.h>
#else
# include <time.h>
#endif /* NOGETTIMEOFDAY */
#ifdef DIRENT
# include <dirent.h>
#else
# ifdef XENIX
# include <sys/dir.h>
# else /* XENIX */
# include <sys/ndir.h>
# endif /* XENIX */
#endif /* DIRENT */
#if (defined(SYSV) && (SYSV < 3))
# include <limits.h>
#endif /* SVR2 */
#ifdef STRINGH
#include <string.h>
#else
#include <strings.h>
#endif
#include "../include/list.h"
#include "../include/tripwire.h"

#if defined(SYSV) && (SYSV < 4)
#ifndef HAVE_LSTAT
# define lstat(x,y) stat(x,y)
#endif
#endif /* SYSV */

```

Figure 2: Sample of header file declarations.

an M4-like preprocessor [8] for the configuration files. Directives such as “`@@define`”, “`@@ifdef`”, “`@@ifhost`”, and “`@@include`” were provided to allow multiple machines to use a configuration file.

We did not use the M4 processor itself, or the `cpp` compiler preprocessor. To do so would have been in contradiction to our earlier efforts to keep Tripwire free of external dependencies and thus, potential vulnerabilities. Furthermore, we could not be certain that these utilities would be present on each platform where Tripwire was to be run. Thus, the decision was made to build a small preprocessor into Tripwire itself.

3.2.2 Lessons learned

Because commands interpreted by the preprocessor reside in the space meant for filenames, a sentinel character must be used to denote the preprocessor directives. This effects how file names

containing the sentinel character must be encoded by the user.

One of us (Gene), in his haste to complete the Tripwire prototype before the start of the Fall 1992 semester, added a literal implementation of the preprocessor functionality suggested in the e-mail sent to us. However, this suggestion, while containing the kernel of a good idea, was an especially poor choice of implementation. The most obvious miscue was the use of a two character sentinel sequence. This is semantically unnecessary — a single character would have been sufficient. However, for compatibility reasons, Tripwire continues to use this sentinel sequence.

During one of our meetings, one of us (Gene) was astonished to discover that the Tripwire parser was not implemented using `lex` or `yacc`. Instead, Gene had decided to hand-code a small parser; he believed that it would take too much effort to generate new files using additional tools. Furthermore, he did not believe that the parser was complex enough to require more than a simple hand-build routine.

However, events proved Gene's estimation to be incorrect. Several small bugs and special cases appeared that led to rewrites of the parser. After discussing the contents of future releases, and after a short discussion with Gene about using `lex` and `yacc`, a machine-generated parser was incorporated to allow these future additions.

3.3 Other surprises: orders of magnitude

3.3.1 Do "static file systems" really exist?

The original Tripwire design and implementation stressed the need to maximize the integrity of the baseline database. As the reports generated by Tripwire are only as secure and reliable as its input data, we stressed that the baseline database be immediately moved to some *hardware-enforced read-only* media to prevent tampering. Allowing the database to be updated in any automated manner seemed inherently unsafe: it could allow an intruder to modify the database and thus subvert the entire integrity checking scheme.

The first group of Tripwire testers perceived this inability to update the database as a serious shortcoming. Their filesystems would undergo small, but frequent changes. This would require them to reinitialize the entire Tripwire database after each change so as to get a clean report. This was tedious, at best. However, the perceived need for running an integrity checker on their systems outweighed their complaints.

The first documented complaint of the inability to make database updates was registered in September 1992, shortly after the beta test period began. We ignored it, as we assumed this was an exceptional case that we would not try to support. After we received a steady stream of similar complaints from testers, however, we realized that those static file systems that Tripwire was monitoring were rarely actually static: files seem to change for many unforeseen reasons.

Shortly before the official November 1992 release of Tripwire, we acknowledged that adding database update capabilities was a practical necessity. A command line interface was provided as a means to update database entries that changed. A more convenient mechanism for database updates was added in a test patch released in May 1993, and was not made official until the December 1993 release.

This capability to interactively make database updates was among the most well-received of enhancements we made to Tripwire. However, adding the change made us uncomfortable because it contradicts the precepts of baseline database security that we so stridently emphasized in our documentation. As such, it took almost one year for us to acquiesce and fully implement a convenient mechanism. That Tripwire is used for so many applications outside of intrusion detection is made possible, at least in part, by the ability to easily maintain consistency between the database and the file system.

3.3.2 You do it how often?

Tripwire is essentially a static audit tool because it detects file changes after the fact. It does this by comparing the file's state against information stored in the baseline database. In our design document, we recommended running Tripwire on a regular basis to support a timely notification of file system changes. We suggested "daily" as a reasonable interval.

One can run Tripwire more frequently and thus reduce the time before noteworthy changes are actually reported. Conceptually, if run frequently enough (e.g., every five seconds), Tripwire can function as though it were a real-time intrusion detection tool. However, this was not an application that we were addressing when writing Tripwire. It was certainly not a mode of normal operation that we thought reasonable.

However, during the testing process, several system administrators complained that Tripwire was running too slowly on their machines to allow file systems to be checked hourly. Because they opted to check numerous signatures for each file in the database, the Tripwire runs were not completed by the time the next Tripwire run started! This motivated the addition of a command line option to selectively skip signatures per invocation.

Here, our expectations of the frequency of Tripwire runs differed by an order of magnitude with those of some system administrator. However, by adding a command line option, we were able to accommodate and support their application of Tripwire.

It is possible that careful optimization of the Tripwire code in a future release would allow sessions to be run with full signature checking every hour. Our original development of Tripwire stressed correctness and portability more than speed, and there are many places where the code could reasonably be optimized.

3.3.3 How large is the database?

A request that the authors received early in the testing process was for database compression. Because we were confident that the database files generated by Tripwire would always remain relatively small, these requests were given little attention. At that time, we still assumed that Tripwire would typically be used to monitor several hundreds of files at most.

A year after official release of Tripwire in November 1992, we started to receive more mail requesting database compression. Users would write about how Tripwire would fill up the entire root partition on many of their machines. One of us (Gene) replied that this was probably a system configuration error on their part; it was inconceivable to us that Tripwire database files would ever grow so large as to impact disk usage. Gene then challenged the user to show how large how the databases were, claiming that databases should be "not that large; certainly no larger than a moderately-sized GIF file."

A refutation came in the form of directory listings of 15 database files, collectively taking up 45 MBytes of disk space. Apparently, many sites have huge archives of files on server machines on which they run Tripwire. Finally understanding this, we conceded that mechanisms for external programs providing services such as data compression and encryption would be provided in a future release. We also changed the database format to use a more compact representation of file signatures and inode values by using base-64 instead of hexadecimal.

Here, our expectations of database sizes were three orders of magnitude smaller than those found at some sites. It may be the case that users requesting data compression a year prior to this incident needed compressed databases, but they failed to provide either Gene with a clear understanding of that need.

3.4 Good surprises

The mark of a good tool is that it is used in ways that its author never thought of.
Brian Kernighan⁴

Potentially less exciting than the stories of intruders being caught, but equally inspiring, are the dozens of stories we have received of sites using Tripwire as a system administration tool. System administrators report having found hundreds of program binaries changed on their systems, only to discover that another person with system-level access had made the changes without following local notification policy.

There has also been one reported case of a system administrator detecting a failing disk with Tripwire. The normal system log reporting the failure was not read very often by the system administrator, but the Tripwire output was surveyed daily.

We used Tripwire to discover a bug in the `patch` program that causes context diffs to be applied incorrectly in a certain special case. This was discovered when the Tripwire test suite failed on a patched file. (See section 4.1.3.)

All these classes of stories further validate the theory behind integrity checking programs. Although the foundations of integrity checkers in UNIX security have been discussed in [3, 4, 7], when Tripwire design was started in late 1991, no usable, publically available integrity tools existed — providing one of the primary motivations for writing Tripwire.

Another application we note uses Tripwire to help salvage file systems not completely repaired by `fsck`, the program run at system startup that ensures consistency between file data and their inodes. In cases where inodes cannot be bound to a file name in a directory, they are placed in the `lost+found` directory and named some (less than useful) number. If a Tripwire database of file signatures is available, this file can be rebound to its original name by searching the database for a matching signature.

Many system administrators use `siggen`, a supplementary program included with Tripwire, as a convenient program for generating commonly used signatures. For example, recent security alerts and patches from response teams such as the CERT and CIAC often contain message digest signatures of patch files; `siggen` can be used to verify these signatures against the announcement.

Because providing a useful tool to system administrators was one of the primary goals of writing Tripwire, the variety of applications of Tripwire outside the domain of intrusion detection has been especially surprising and satisfying for us. We are still collecting other stories of novel use of the Tripwire package.

4 Patch, package, and release (repeat as necessary)

We have found that the process of building and releasing new versions of Tripwire is one of the most difficult tasks we have faced in the last three years.

4.1 Providing a rigorous test suite

To ensure the correct operation of Tripwire on platforms to which we do not have access, we include a test suite with the distribution. The test suite was originally one shell script that exercised all the signature functions. It would compare the signatures of all the files in the distribution against those included in a test database. As a side-effect, this would ensure the integrity of the files in the distribution.

As time has gone on, we have expanded the self-tests performed. Tripwire currently has a suite of seven shell scripts that also test for correct functioning of various Tripwire operations including database updating, reporting, preprocessing, and using alternate modes of input.

⁴Brian Kernighan has said this, in one form or another, in several of his presentations and written works. This particular version was in private e-mail to one of us in response to a citation request.

4.1.1 When tests know what files are supposed to look like

Generating signature functions that operated correctly across machines of differing architectures (i.e., big- vs. little-endian) proved exasperating; several of the releases bore patches to make the signature routines work on yet more architectures. In part, we can blame this on the fact that we incorporated existing versions of some of these signature routines as coded by other authors. This was done to avoid transcription errors. It is also traceable to the notion that it would be less error-prone than if we coded the routines ourselves. Unfortunately, we discovered too late that the original code was not written with portability in mind.

This rash of signature function errors motivated us to write a test script to ensure the correct generation of signatures of known files — namely, the Tripwire source files. This test evolved into a general Tripwire run, checking the source file signatures against those stored in an included test database.

This test script has been appreciated by our Tripwire users because it demonstrates Tripwire operation on a smaller (if contrived) scale. Users have commented that it is gratifying to see their modified Makefile and configuration files being reported as changed. However, having a test script that knows about all the files in the distribution has required a far more regimented and structured procedure for building Tripwire releases than we anticipated.

For instance, consider what happened when the first Tripwire patch distributed to testers changed many files. After testers applied the patch, they almost universally reran the test suite (an action that we did not anticipate). Not surprisingly, Tripwire reported all patched files as having changed. This was disconcerting to the testers, and they requested that all future patches install new database entries in the test script.

This new requirement of ensuring database consistency complicated the process of patch and release generation to such an extent that the next three patches released were incorrect in at least one way. Consider the process for putting together a patch release. First, because “checking in” a file under RCS control changes the contents of the file (the RcsId tag is incremented), all files must be first checked in. Next, as the Tripwire test database should be generated using the most recent version of Tripwire, a new Tripwire executable must be compiled. After compiling the sources, the resulting executable is used to generate a test database. Completing this, the patch is then generated (including a patch for the database file). Next, the distribution and patch is moved to a clean directory, compiled, and then tested by running the test script on a variety of platforms. If it completes correctly without any errors, the database is checked in and the entire patch is regenerated.

On a Sequent Symmetry (with 16 MHz 80386 processors), this entire patch build process takes almost thirty minutes. The high iteration time further frustrated a process that was difficult to do correctly.

4.1.2 It's hard to get right

The story of how the first several patches released during the test period is presented here to motivate our decision to fully automate the process of release generation.

Before any patch was released to testers, Gene would send the (hand assembled) completed patch to the other Gene, who would then test it on a number of different machines available to him. Although this first patch applied correctly, the package failed on a certain machine, necessitating a change to the Tripwire source. A new patch was assembled and again sent to Gene, who then reported that the patch failed to apply. Over the course of the next two days, Gene would send six more patches that would also fail for one reason or another, necessitating another build. This process was typical, and so prone to error that Gene insisted on sending out only those patches that he had personally verified.

However, mistakes still occurred. One time, some miscommunication resulted in Gene sending

out a patch that was not “checked in” under RCS control. Because of this, subsequent patches until the next official release had to be hand-edited (correcting all the incorrect RcsId tags) so they would apply properly. To fully understand the scope of this problem, consider that the state of Tripwire sources held by users no longer mapped to any files in our source tree. Checking in the distributed files into our source tree would change their RcsId tags, making all future patches to these files fail to apply correctly.

Many of these problems occurred again when working on the second patch release. This motivated the writing of 440 lines of shell scripts in seven files to automate the generation of patches. As we write this, neither Gene knows how to manually generate a patch release without these scripts.

As each “last minute change” that we try to squeeze in right before a patch release necessitates a full rebuild, much time could be saved by using the Tripwire “database update” command to alter only the one changed entry. However, so convoluted is the entire procedure of generating distributions and patches that Gene has not risked breaking the scripts by touching them.

4.1.3 Other sources of problems

One patch release uncovered a bug in the patch program itself that would misapply the output of a contextual “diff”. The problem was discovered when a user called one of us (Gene) on the telephone, alarmed that the Tripwire test script reported a corrupted header file. Inspecting the file revealed that a newline had been incorrectly inserted by patch.

A workaround was provided by hand-editing the patch file so that the output file would match the intended file.

4.2 Growing better test suites

Since the initial release of Tripwire, seven more test scripts have been added to the Tripwire distribution. The addition of these test scripts was motivated by the desire for canonical test cases in the Tripwire distribution for bug reporting purposes. While tracking a few persistent problems we realized that we needed some method to test all Tripwire features on each new architecture. The result was our test suite.

The entire test suite on a Sun workstation now takes as long as twenty minutes to execute. Yet, to the best of our knowledge, running the test suite is the first thing that most users do after building Tripwire. One of us (Gene) expected that users would find it too time-consuming and use Tripwire without running the test suite; the other Gene knew from experience that the actual behavior was the likely case.

The testing environment scripts for the functional testing scripts were originally written in 70 lines of Perl. To ensure that sites without Perl can run the Tripwire test suite, the test suite was rewritten as 160 lines of Bourne shell script with functions. When one of us (Gene) learned that many older machines do not support inline functions in Bourne shell script, it was rewritten as a 240 line “classic Bourne” shell script that ran four times again as slowly. Even with this change, some sites report system oddities that prevent the script from working (including malfunctioning test commands).

4.3 Good tools give great mileage

A number of software tools added to the quality of Tripwire code. We made extensive use of several publically available tools in writing Tripwire. Those that we feel added significantly to the quality of Tripwire code are discussed below.

4.3.1 Pedantic compilers

A large portion of changes to the Tripwire code during the Tripwire testing period was from users compiling Tripwire on SGI workstations. More than any of the compilers used, those pedanti-

cally rejected any marginally unhealthy coding convention. The changes necessary to get Tripwire to compile on SGI machines would be invariably sent to us, where it would get merged into the Tripwire sources. Tripwire now “lints clean,” thanks to an ambitious programmer in Australia who sent us a 70 Kbyte patch file in October 1992.

4.3.2 ANSI C prototypes

Using ANSI-style C prototypes saved the authors countless hours of debugging. By enforcing argument consistency in function calls, numerous errors that would have resulted in non-obvious program misoperation generated errors at compile-time.

However, unlike `lint`, ANSI compilers must have function prototypes defined to catch these argument calling errors. Therefore, prototypes to all functions in Tripwire are stored in one header file. Because of this, changing even one prototype in this header file has a side effect of rebuilding all the Tripwire object files because of our Makefile-driven builds. (Having multiple prototype header files was rejected by Gene as being too ugly.)

4.3.3 `cproto`

The easy generation of ANSI C prototypes is made possible by the `cproto` tool by Chin Huang⁵. This program uses a partial C grammar to automatically generate correct ANSI prototypes. The resulting header files will be appropriately wrapped with macros so K&R C compilers will interpret them as conventional function declarations.

4.3.4 Source control

Another tool used extensively by Gene is the CVS (Concurrent Versions System) tool by Brian Berliner [1].⁶ CVS is a front-end to the well-known RCS utilities [14], providing useful “release revision” abstractions beyond the file-by-file revisions afforded by RCS. Among other things, CVS allowed for the easy generation of patch files against any previously released version of Tripwire (modulo the contortions necessary for the self-test database).

4.3.5 Commercial tools

Tripwire development also benefitted from the use of two commercial products: CodeCenter by CenterLine Software and Purify by Pure Software. CodeCenter (formerly known as Saber-C) provides an interpreted C environment, where even the most seemingly benign errors (e.g., function argument misuse, potentially erroneous C statements, etc.) are reported at compile- and run-time. One of the most subtle bugs in Tripwire (the error condition described in section 5.2) was eventually fixed with the aid of CodeCenter.

Purify is a library that is linked at compile-time. When the program is run, it detects memory access errors and instances of memory leaks. It not only helped detect numerous errors before each of the later releases of Tripwire, it also allowed for the elimination of conditions where memory was not being reclaimed. (It also detected an error in the `libc` library being shipped with SunOS 4.1.3: calling `ctime()` results in a warning about writing to an illegal memory location.)

Using commercial tools involved more restrictions than the other tools we used, mostly because of licensing and platform problems. For instance, Gene did not have accounts on machines where these packages were installed. So, the other Gene would run the programs on his machine, and then send the output to Gene via e-mail. (Gene appreciates that he was not ever required to submit punch cards. He was, however, forced to read stack traces, without line numbers, generated by Purify.)

⁵Chin Huang can be reached at chin.huang@canrem.com. His `cproto` program can be found in volume 31 of the `comp.sources.misc` archive.

⁶Brian Berliner can be reached at Brian.Berliner@Sun.COM. His CVS program can be obtained from the GNU archives via anonymous FTP at [prep.ai.mit.edu](ftp://prep.ai.mit.edu) in `./pub/gnu`.

4.4 Finding veteran guinea pigs

The Tripwire development process has been greatly aided by the keen interest generated in the system administrator community. The 200+ beta testers who assisted the authors in the summer and fall of 1992 not only tested Tripwire, but suggested the addition of significant enhancements that undoubtedly have helped lead to its widespread use today.

Since the official release, Tripwire has benefitted from similar groups who tested patches for the five test releases. Generally, we have limited these groups to under fifty people, so as to reduce the number of repeat bug reports to a manageable level and to simplify the distribution process. Invariably, a number of bugs are found and a number of features are added at the last minute.

It is interesting to note that very few members involved in Tripwire testing opt to volunteer for the next callout. It is gratifying that there always seem to be an eager group of Tripwire users happy to take part in the release testing given a moment's notice. At the same time, we wonder what has dissuaded the previous testers from volunteering again.

The authors feel very fortunate to have had such an eager and talented group of testers. There is no doubt in our minds that they have made a significant and substantial impact on the direction of Tripwire development. We have tried to credit all of them in the documentation shipped with Tripwire to recognize their contributions.

4.5 Receiving gifts from the code elves

Tripwire has benefited from generous and helpful code reviews provided by people who both Genes respect highly. Code changes received from these people include hash table speedups, base 64 routine speedups, and some inevitable comments on undesirable coding practices.

We have received two patches of almost 80 Kbytes in length that have added significantly to the Tripwire tool. The first was the previously mentioned patch that made Tripwire generate only minimal output when checked by `lint`. The second patch we received from another person in January 1994 added handling for symbolic links — a previously documented weakness of Tripwire. The patch also fixed a few lurking bugs, and offered useful comments on design issues untouched since the summer of 1992.

However, not all of the contributions we have received are so spectacular in their contents. One of us (Gene), having been made more wary of blindly implementing user suggestions, has patches that do not produce compilable files, produce clearly undesired side-effects, destroy data structure assertions, or are just wrong. We both usually write back, thanking the users for their time and consideration.

4.6 Lingua franca, s'il vous plait, bitte?

The test distributions of Tripwire included a tentative design document that one of us (Gene) developed before starting actual coding. This document served to inform testers of the design goals of Tripwire — issues that the incomplete Tripwire program conveyed far less directly. The official November 1992 release of Tripwire did not include this document, being unfit for widespread distribution or scrutiny, but promised that one would be “made available soon.”

Hundreds of requests for this document were saved until November 1993 when the design document was finally completed. However, unlike the original document that was written using `troff` and the almost ubiquitous “ms” macros [13, 12], the new document was written using the `LATEX` formatting system [11].

The fact that some users would be unable to generate printable text from a `LATEX` source file made its inclusion in the Tripwire distribution problematical. We eventually included the design document formatted in PostScript. We resolved the problem for the lowest common denominator (i.e., those without PostScript printers) by making the document available as a technical report via surface mail.

The L^AT_EX source file without diagrams is 52 Kbytes. The generated PostScript file is 223 Kbytes, and adds to an already already large 7 Mbyte distribution.

5 What we still don't know how to do

5.1 Running Tripwire on the Sasquatch Kumquat Mark VIIa/MP

Tripwire has proven to be highly portable, successfully running on over 28 UNIX platforms. Among them are Sun, SGI, HP, Sequent, Pyramids, Crays, Apollos, NeXTs, BSDI, Lynix, Apple Macintosh, and even Xenix. Configurations for new operating systems has proven to be sufficiently general to necessitate the inclusion of only eight example `tw.config` files.

However, potentially challenging situations result when we receive requests from system administrators asking for help compiling Tripwire on machines that neither of us have ever heard of. In one case, this was a machine only sold in Australia and shipped with incorrect system libraries. Other instances included an especially ignoble machine that has not been sold since 1986 (predating college for one of us), and numerous machines with non-standard compilers, libraries, system calls, and shells.

In all but two cases (of the last variety), we have incorporated changes in Tripwire sources to accommodate these machines. In most cases, there has been a sufficiently large group of system administrators with similarly orphaned machines who put together a suitable patch to allow correct Tripwire compilation and operation.

It is interesting to analyze the time needed to fully support a configuration. Full support for Sun's new Solaris operating system was added two months after the initial Tripwire release. A workaround for the two aforementioned Australian machines was released six months after the problems were first reported. However, some Tripwire users running machines from a large workstation vendor continue to be unable to find a compiler that correctly generates a Tripwire executable that passes the entire test suite; investigation has determined that this is because of non-standard and broken compilers and libraries on those platforms.

5.2 Invalidating all those old versions

We often get mail from users running old versions where the bugs they report have been fixed in a later release. In one case, this was a patch released in 1993 including a section of code that checks for a rare boundary condition. Having detected such a condition, Tripwire would print a warning banner:

```
added:  -rwxr-xr-x root      16384 Jul 23 13:44:55 1992 /usr/etc
### Why is this file also marked as DELETED?
### Please mail this output to (genek@cc.purdue.edu)!
```

Although an important bug was discovered through this invariant testing and a corrective patch distributed, the authors continued to receive daily reports of this bug two months after the fix was released. The Tripwire distribution available through our local FTP server includes the fixes for the problem, but older versions are also available at numerous mirror FTP sites around the world, as well as through any `comp.sources.unix` archive - information servers (e.g., `archie`) can lead users to out-of-date sources.

There is no mechanism for invalidating the older versions of Tripwire that still reside at these FTP servers. One year after the corrective patch, e-mail about this bug has finally abated. However, an e-mail dated February 17, 1994 is evidence that these older versions are still present on some FTP servers.

5.3 Maintaining a useful mailing list

To assist in the dissemination of information on Tripwire, a mailing list was established. Intended for the fielding of Tripwire questions, answers, and usage, the authors expected the mailing list to be very active, reflecting the traffic on certain USENET newsgroups such as `comp.unix.security` and `alt.security`.

However, throughout its lifetime, the mailing list remained mostly quiescent, disturbed only by misaddressed requests to subscribe to the mailing list. When bursts of correspondence did take place discussing the addition of certain features, a flurry of electronic mail from users clamoring to be dropped from the mailing list resulted. The electronic mail traffic asking to be dropped from the mailing list always dwarfed the amount of traffic devoted to Tripwire discussions.

As a result, we have discontinued the mailing list, although personal correspondence with regards to Tripwire to both Genes remain high. However, not all users use personal e-mail correspondence to communicate with the Tripwire authors. There are a surprisingly large number of people who attempt to contact us by posting a USENET article, sometimes to clearly inappropriate groups. This strikes us as the equivalent of walking into a grocery store and yelling, "Could I speak to the person who invented Charmin?" Nonetheless, we note that we have personally counted at least four instances of this in February 1994 alone, and that we have replied to those messages — can we not conclude their methods sometimes work?

5.4 Keeping track of all that documentation

With the exception of an interactive database update mode, the user interface to Tripwire is via the command-line. Ensuring consistency between available command-line options and all associated documentation seems straight-forward, but has been increasingly tedious as more options and features are added.

Tripwire is documented primarily through its manual page and its usage message. However, as new features are added, the manual page is usually the last to be updated. Recently, some new features have instead been documented in a `README` file, and even a `WHATSOEVER` for those items that must stand out among those features described in the `README` file. (Considerable discipline was required to restrain from then creating multiple `WHATSOEVER` files!)

Maintaining consistency between the program itself and its usage message requires considerable discipline, especially considering the lack of locality between the program code, usage message, and manual page. Several tools might help to automatically generate manual pages and usage messages from the program code, but the authors have not used them. Given the tedium of these issues, further Tripwire development would do well to investigate these tools.

One of us (Gene) recently reviewed the source code to check his commenting consistency. There were many instances of old comments lingering about, despite the pertinent code having been deleted. There are still several sections of code `#ifdefined` out, but not deleted for fear of deleting something actually important.

5.5 Handling those bug reports

Associated with the problems of handling bug reports are their tracking. There have been a number of tools developed to track bugs. However, in the case of Tripwire, bugs have been tracked exclusively with mailers.

With very few exceptions, bugs are reported via e-mail, having been sent to either Gene or both Genes. Because of the informal nature of our bug tracking, there have been several bug reports lost or misplaced. Consequently, the bug reports that were sent to both Genes have had a better response rate. During our meetings, one of the topics always discussed was whether replies have been made to all people inquiring about certain Tripwire functionalities or bug reports.

5.6 Working with the other Gene

Many of the design misdecisions presented earlier came about because of the lack of contact between both Genes. The combination of Spafford's travel schedule and Kim's lack of diurnal habits made regular contact difficult. When Kim was avoiding Spafford for two months because of diminished progress with Tripwire, even this contact disappeared. Several times, Tripwire progress languished for almost an entire semester, but then experienced frantic progress in the final weeks (or sometimes, days). Not surprisingly, these conditions do not consistently produce good code or good decisions.

In many cases, the perception of the state of Tripwire held by each Gene would diverge, meeting only when enough misunderstandings prompted a long meeting to discuss the true state of Tripwire. This would often occur when both Genes would reply independently to an e-mail request and notice that their answers disagreed.

5.7 Moving the source base

Another consequence of working as an undergraduate student involved in internship programs was the lack of stable resources (e.g., workstation, disks). The Tripwire source tree has resided on six different machines at four institutions over three years. Although this has made the Tripwire distribution tools portable, considerable time and effort has been invested in portions of Tripwire that users will never see.

5.8 Purging American biases

We faced a surprising aspect of portability when we received a request from a user in Australia. He complained that Tripwire would always ignore his command line invocation: `"tripwire -initialise"`. After a day of frustrations, he finally sent one of us e-mail asking us to support non-American spellings of "initialize."

5.9 Other problems of scale

By most measures, the Tripwire project has surpassed our expectations. Among them is the amount of time. To date, almost three years have passed from the start of design study to product delivery and continuing support. Both Genes are relieved that this "one semester project" was not a graduation requirement for Gene Kim.

5.10 Other lowest common denominators

We continue to be amazed how low the "lowest common denominators" for UNIX systems are. Less charitably, but perhaps equally valid, is the same observation for users.

We have received a number of e-mail messages that have titles similar to "Tripwire setup allow read access?" with messages that apparently contain a question, but no discernible way to glean what the question might be. We have received e-mail asking how to get Tripwire running on a machine, and the only data we are provided with is the name of their machine and that "Tripwire doesn't work." At least one inquiry was received totally in a foreign language not spoken by either Gene. Responses seeking clarification of these requests usually bounce.

Both Genes are very proficient in deleting e-mail.

6 Conclusion

In this paper we have described our initial design goals of Tripwire, what we ended up with, how we got there, and discussed some of the surprising issues that arose on the way.

Some of these challenges we handled well, and others not quite as well — which is probably the more interesting of the two. Among the factors to which we attribute these mishandlings are gross misestimates of scale (how Tripwire was used, how often, how large, etc.), miscommunication,

inexperience, and semester deadlines.

However, in the past three years, we have substantially improved our ability to put out releases that are free of errors introduced by procedural mistakes and awkward compatibility issues. We have a rigorous test suite that exercises major functions and performs regression testing, and we have established a testing procedure that has allowed us to find most bugs before the majority of users even get access to Tripwire.

However, other issues remain that are quite separate from applications development that we would eventually like to address. Among them are supporting the most obscure (if not baroque) UNIX machines, controlling and invalidating copies in out-of-date FTP archives, and providing a good forum for discussion of Tripwire issues.

Perhaps the aspect of Tripwire development of which we are most proud is that the majority of Tripwire users remain completely unaware of the problems we have had to overcome. Since November 1992, we have been able to provide a tool for system administrators that is freely available, widely distributed, applicable, and well-supported, and that addresses a significant security need.

7 Availability

Tripwire source is available at no cost.⁷ It is available via anonymous FTP from many sites; the master copy is at URL "<ftp://ftp.cs.purdue.edu/pub/spaf/COAST/Tripwire>". The source and several papers about Tripwire can be accessed via WorldWide Web at URL "<http://www.cs.purdue.edu/homes/spaf/coast.html>". Those without Internet access can obtain information on obtaining sources and patches via e-mail by mailing to tripwire-request@cs.purdue.edu with the single word "help" in the message body.

We regret that we do not have the resources available to make tapes or diskette versions of Tripwire for anyone other than COAST Project sponsors. Therefore, we ask that you not send us media for copies – it will not be returned.

References

- [1] Brian Berliner. CVS II: Parallelizing software development. In *Proceedings of the Winter Conference*, Berkely, CA, 1990. Usenix Association.
- [2] Matt Bishop, November 1993. personal communication (11/6/1993).
- [3] Vesselin Bontchev. Possible virus attacks against integrity programs and how to prevent them. Technical report, Virus Test Center, University of Hamburg, 1993.
- [4] David A. Curry. *UNIX System Security: A Guide for Users and System Administrators*. Addison-Wesley, Reading, MA, 1992.
- [5] Paul Dickson. *The New Official Rules*. Addison-Wesley, 1989.
- [6] Daniel Farmer and Eugene H. Spafford. The COPS security checker system. In *Proceedings of the Summer Conference*, pages 165–190, Berkely, CA, 1990. Usenix Association.
- [7] Simson Garfinkel and Gene Spafford. *Practical Unix Security*. O'Reilly & Associates, Inc., Sebastopol, CA, 1991.
- [8] Brian W. Kernighan and Dennis M. Ritchie. *The M4 Macro Processor*. AT&T Bell Laboratories, 1977.
- [9] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: A file system integrity checker. Technical Report CSD-TR-93-071, Purdue University, nov 1993.

⁷It is not "free" software, however. Tripwire and some of the signature routines bear copyright notices allowing free use for non-commercial purposes.

- [10] Gene H. Kim and Eugene H. Spafford. Experiences with tripwire: Using integrity checkers for intrusion detection. In *Systems Administration, Networking and Security Conference III*. Usenix, April 1994.
- [11] Leslie Lamport. *L^AT_EX: User's Guide & Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [12] M. E. Lesk. *Using the -ms Macros with Troff and Nroff*. AT&T Bell Laboratories, 1976.
- [13] Joseph F. Ossana. *Troff User's Manual*. AT&T Bell Laboratories, 1976.
- [14] Walter F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering*. IEEE Press, September 1982.
- [15] David Vincenzetti and Massimo Crottozzi. ATP anti tampering program. In Edward DeHart, editor, *Proceedings of the Security IV Conference*, pages 79–90, Berkeley, CA, 1993. USENIX Association.

Design, Distribution and Management of Object-Oriented Software

Arindam Banerji, David Cohn, Dinesh Kulkarni
Distributed Computing Research Laboratory
University of Notre Dame
Notre Dame, IN 46556
axb@cse.nd.edu

Abstract

The promise of object-oriented software has been somewhat dimmed by the continuing need for source code familiarity to realize the goals of code-reuse and manageability. Software design has been hampered by the infeasibility of predicting all possible circumstances of use. Composing applications out of reusable components has remained a myth, primarily due to limitations of the simplistic shared library model. This paper proposes a three-pronged attack on these limitations of object-oriented software in the context of C++. A flexibility framework which facilitates the extension and modification of software without recompilations or source-code familiarity is described. Partially resolved loadable subclasses that may be distributed as reusable units for type-safe application composition. Specific programming guidelines which allow implementors to create software that may be fine-tuned at run-time according to application characteristics is described. Thus, the paper proposes a set of tools, techniques and guidelines that can facilitate the construction of application frameworks.

1. Introduction

Object-orientation has not lived up to its promise of promoting code-reuse and simplifying maintenance. Software ICs [Cox, 86] that were touted as solutions for expediting the development and maintenance of software, have failed to materialize. Here, we discuss some of the problems related to this issue and explore a possible solution in the context of C++, the most widely used object-oriented language.

We feel that a major roadblock on the way to easy maintenance and reusability, is the lack of *extensibility*. The designer and the initial implementor of a class library cannot predict *all* possible future uses. Hence, modifications and extensions of existing software typically requires an in-depth familiarity with its source code. However, if some of the normally *hidden* aspects of a software component's design and implementation are exposed in a disciplined fashion, the component can be extended without dependence on source code familiarity. We contend that inclusion of these facilities for extension and flexibility should be a basic design principle for object-oriented software.

Flexibility refers to the ability of a client of a software to tune the implementation of an object to suit client-specific needs. For example, adjusting the buffer size in a communication protocol implementation or changing the paging policy from least-recently used to most-recently used (for sequential reads) are examples of flexibility at work. Extensibility on the other hand entails major behavioral modifications; e.g. replacing a protocol implementation with a better implementation that provides additional functionality by adding methods to the previous interface. It is important to emphasize that these two properties - flexibility and extensibility should be available at run-time, not just design and implementation time.

Extensibility would simplify distribution of class libraries. The need for releasing source-code would be substantially reduced; binary distribution would be sufficient in most cases. In addition, extensibility would make it possible for a user to dynamically integrate third party extensions to a particular product, thereby composing a tailored version of a standard application. This will safeguard the software vendor's investment without constraining clients who want to customize or enhance software. Moreover, it allows third-party vendors to sell customized extensions, much like Software ICs.

Flexibility and extensibility can be envisioned at several levels. Replacing a part or all of an implementation, binding to a different interface or even extending an interface, are all desirable for binary distribution without unduly constraining the client of the libraries. For example, a comprehensive communication software could choose between TCP/IP or NetBIOS interfaces, replace a UDP implementation or even extend the socket interface to support mobile communication. On another level, basic support technology for run-time subclass loading allows for dynamic extensions to existing software. In fact facilities for making such changes can be neatly organized into a framework.

Frameworks have been used as common units of reusable software [Deutsch, 89] [Wirfs-Brock, 90]. A framework is a set of collaborating class hierarchies that provides a certain Application Programming Interface (API) and uses a certain service-provider interface (SPI). For such application frameworks, extensibility could mean using a variant of the SPI, providing a variant of the API or modification of behavior and/or implementation. But there is a common thread that runs across these seemingly different aspects of flexibility. This thread can be captured in a framework for extensibility and flexibility that provides the basic mechanisms and a structure for tailoring an application framework. This framework forms the basis for application or domain-specific frameworks, and is the focus of this paper.

The proposed framework adds structure beyond the essential features of encapsulation and inheritance. It changes the black-box nature of components by providing facilities for exposing their implementation and for dynamically changing their behavior in an incremental fashion. It does so through a clean separation between an interface and its implementations on the one hand, and the interface and an associated instance on the other. It also uses systematic indirection for flexible binding and dispatch of alternate method implementations. A prototype of the basic components of the framework has been developed in AIX 3.2 using a custom variant of AT&T C++ translator. Sophisticated dynamic linking facilities constructed using AIX linker, allow for loading and binding of subclasses to running C++ programs.

The next section further explains the basic mechanisms for flexibility. It is followed by a section on programmer's view of extensible software and guidelines for extensibility. Section 4 discusses a programmer's view of a prototype implementation. Section 5 reviews some of the related work.

2. A Framework for Extensibility and Flexibility

As indicated in the preceding section, extensibility and flexibility must be supported from ground up rather than as add-on features or ad-hoc facilities in a particular application. This requires that the basic object model should itself address these issues and provide primitive facilities in the form of a generic framework which can then be used in different applications.

2.1. Basic Object Model

The basic object model, described in [Banerji, 93], envisages a clear separation between instances, interfaces and implementations as shown in Figure 1. The instance-interface separation allows multiple instances of an interface to change their behavior independently, and, since interfaces are explicit entities, they can be changed.

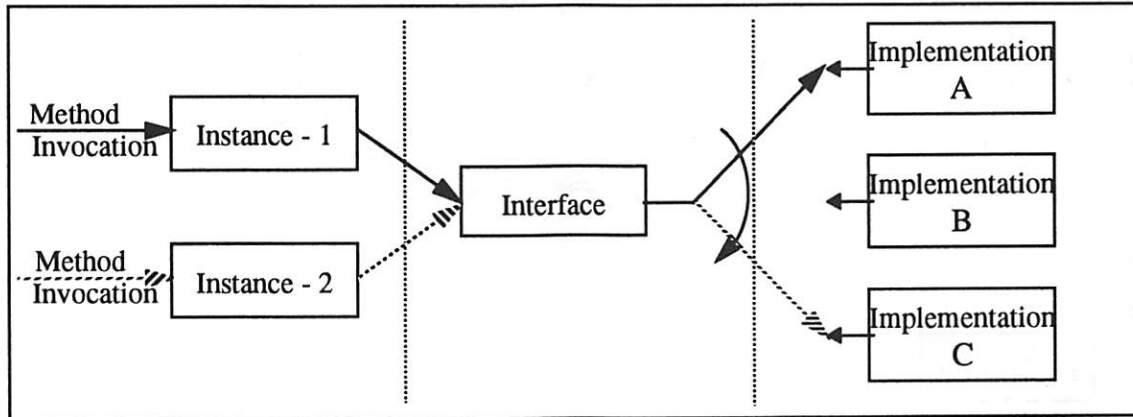


Figure 1 - Separating Instances, Interface and Implementations

Separation of interface from implementations ensures that multiple implementations for a particular interface can co-exist. This allows a client to choose a particular implementation at run-time. Initial implementors would build at least a default implementation, and users could add additional ones. One of the goals of such a separation is to ensure that classes representing interfaces depend upon very few symbols from classes representing implementations. This is especially helpful when implementations are loaded dynamically (as will be seen later) and symbol-binding has to be performed.

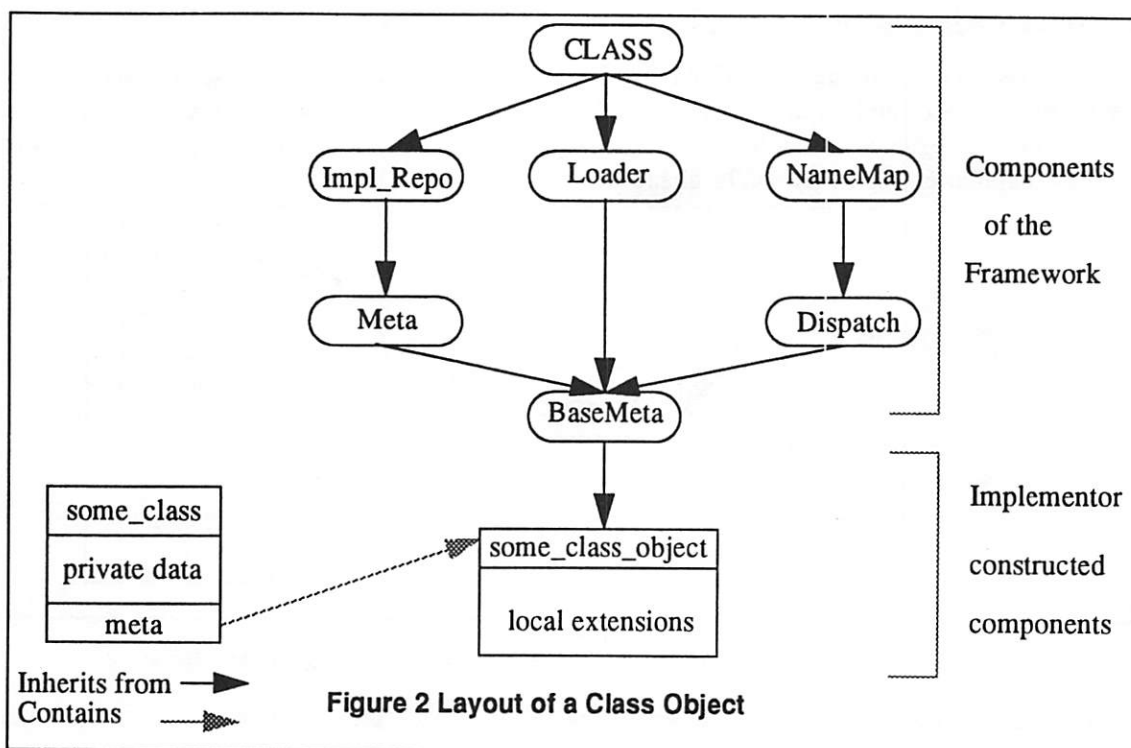
The initial implementation of this object model, as realized by the extensibility framework, currently provides only the interface-implementation separation. The mechanisms for providing this kind of a separation are discussed in Section 5 on programming guidelines.

2.2. Framework Components and Techniques

The framework utilizes indirection as a key mechanism for providing flexibility. Indirection in function dispatch ensures that implicit entities such as name-resolution and member function lookup can be directly controlled by both initial implementors and client-programmers. This control is usually realized in two forms. Function-names not known at compile time can become call-targets, allowing functions and classes that are loaded at run-time to be used directly. Preprocessing and postprocessing code can be dynamically attached to any member function, permitting the addition or deletion of pre-condition and post-condition functions on the fly.

Indirections are typically hosted in *class-objects* [Goldberg, 89]. These objects may be associated with any class for which extensibility is desired. Class-objects, also known as *meta-objects*, provide a run-time representation of normally implicit class implementation aspects. Implementors can construct class objects by specializing the facilities provided by the framework. The relationship between the implementor-constructed class-objects and the framework components is shown in Figure 2.

The upper part of the figure shows the set of collaborating class-hierarchies that are the components of the framework. These are combined by multiple-inheritance into the class **BaseMeta**.



Class objects are then direct descendents of the class **BaseMeta**. They thus inherit the indirections provided by the framework.

These indirections are of two kinds: constructor-call and dynamic-dispatch. Constructor-call indirections allow the instantiation of classes whose names are not known at compile time. Thus, dynamically loaded classes identified by some characteristic string, such as their name, may be instantiated by client-code. Dynamic-dispatch indirection, on the other hand, allows the target of a member-function call to be derived from a variety of parameters and conditions, not just from the type-specification of the function. This allows client code to manipulate exactly how the target of a named function call is resolved. Constructor-call indirection allows dynamic loading of new function implementations and dynamic-dispatch indirection allows clients to determine when these implementations are used.

These two kinds of indirection can be combined to facilitate the management of object-oriented libraries in two very special ways:

- Dynamically add or change implementations for a particular interface.
- Dynamically extend the interface provided by a library.

These changes can be made *without access to the source code*, providing true software reusability.

2.3. Constructor-Call Indirection

Dynamically loaded classes, enabled by indirections in constructor calling represent a unique problem since their names and the names of their member functions may not be known at compile-time. One approach is to only dynamically load subclasses of existing classes with matching function names. These functions must have initially been declared virtual and calls are then made through virtual function tables, ensuring type-safety. Of course, this limits new subclasses to using only previously known function names.

This still leaves the question of constructors, which in C++ cannot be declared virtual. Constructors of loadable subclasses need to be called indirectly to avoid unresolved externals during

compilation. The classes `Meta` and `Impl_Repo` in Figure 2 implement this functionality. The details of the class `Meta` are shown below in a sample code fragment. The class `Meta` essentially supports three pieces of functionality:

- Functions `register_class_obj` and `unregister_class_obj` that allow the class-object of a loaded subclass to announce itself as available or unavailable as a possible target of constructor-call indirection.
- Overloaded function `instantiate` that actually redirects constructor calls.
- An instance of `Impl_repo` that acts as a database of available subclasses which can be targets of constructor call indirection.

The following code fragment shows the key features of the class `Meta`.

```
class Meta: public CLASS {
public:
    // constructors and destructors
/*
    In the following code ImplSpec is used to specify some characteristic
    of the implementation. In the simplest case, it may simply be a string
    containing the name of a class.
*/
    virtual void register_class_object(ImplSpec *,...);
    // function that allows the addition of the class-object
    // of a subclass1 to the repository.
    virtual void unregister_class_object(ImplSpec *,...);
    // removes the entry for the class-object of a sub-class
    // from the repository.
    virtual CLASS *instantiate(ImplSpec *);
    // an indirect constructor called with some form of a
    // of a specification of which particular subclass
    // needs to be instantiated.
private:
    static Impl_Repo *repository;
}; // specification of the Meta class.
```

As expected, loadable subclasses are associated with class-objects. Global instances of these class-objects allow for automatic registration with the class-object for the base class, through calls to static constructors. For example, if `foo` and `fooMeta` are respectively the superclass and its associated class object and `childof_foo` and `childof_fooMeta` are the derived class and its associated class object, then the following happens during static initialization:

```
childof_fooMeta childof_foo::meta = new childof_fooMeta;
```

This call to the constructor of `childof_fooMeta` causes it to register itself with `fooMeta`, as the class object for `childof_foo`. At this point, when instantiation requests for `childof_foo` are sent to `fooMeta`, they are automatically forwarded to the `instantiate` in `childof_fooMeta`. This `instantiate` function, in turn calls the constructor of `childof_foo`. If there are additional parameters that need to be passed to the constructor of `childof_foo`, then the code gets more complicated, but the principle remains the same.

1. Subclass here refers to the subclass of the class that the class that `Meta` is associated with. This, if `foo` is associated with `fooMeta`, a specialization of `Meta`, then the subclass here refers to `childof_foo`, a class derived from `foo`.

2.4. Dynamic Dispatch Indirection

Quite often, the ability to add or substitute class implementations is not sufficient. Certain circumstances require the interfaces themselves to be changed. When recompilation is possible, inheritance is used to create a modified class. However, if recompilation is not an option, a mechanism for adding member functions to existing interface classes is required. The components **Dispatch** and **NameMap** provide this functionality. The main features of these components are:

- The functions **add_intf** and **remove_intf** which allow the addition of member functions to the interface.
- An instance of **NameMap** that maintains the list of all the member functions added in this way.
- The function **generic_func** which indirectly calls the newly added member functions.
- Utility functions such as **add_post** and **add_pre** that add preprocessing and postprocessing code to member function.

The implementor creates a set of function macros, which map newly added member function calls to calls to **generic_func**. This function, in turn calls the actual function in the implementation, ensuring that no unresolved externals are generated during compilation.

In addition to the infrastructure described above, the **Loader** class provides control over loading of new code. This class gives clients of extensible software libraries an interface to add implementations at run-time.

3. Loadable Subclasses and Run-Time Loading

Much of the flexibility and reusability of the work described in this paper is gained from the ability to dynamically load and bind code. This entails the construction of special shared libraries for subclasses and a dynamic loading library which implements much of the functionality provided by the class **Loader** in Figure 2. The description that follows is specific to AIX 3.2 on the RS/6000s and the AT&T cfront compiler.

In order to create the appropriate shared libraries, the following steps need to be taken:

- Compile the source for the loadable subclasses and their associated class objects.
- Create a list of all symbols that are to be exported by this library.
- Create a list of all the symbols imported by this library, that should actually be resolved at run-time.
- Generate uniquely named library initialization and termination code and compile them.
- Use custom cfront compiler driver to link object modules from first and fourth steps into a shared library that has uniquely named arrays for holding static constructors and destructors.

Almost all the steps are completely automated, through generic makefiles and specially designed tools. The implementor need only specify the names of the classes that are to be exported by the library so created. Everything else is completely automated.

The run-time loading library dynamically links in and loads C++ libraries into running programs. At present, it only supports the XCOFF executable file format of AIX 3.2. The interface supported by the library mimics the SUNOS run-time linker calls of **dlopen**, **dlsym** and **dlclose**. During the call to **dlopen**, all unresolved externals within the loadable library, created as shown above, are bound to the appropriate symbols of the running client program. The **dlopen** and **dlclose** calls also ensure that the entry-points for calling static constructors and destructors, get called automatically.

It is very important to point out the difference between loadable subclasses as discussed here and ordinary shared libraries. Loadable subclasses do not preclude the type-safety provided by the C++ language. Moreover, many interesting characteristics of C++ such as virtual functions and

inheritance do span across dynamically loaded subclass components. The features provided by these linking facilities would enable third party vendors to sell value-added subclasses for existing products, without being forced to ship any of the original product's class libraries. This ameliorates licensing problems and opens up possibilities for true pluggable software ICs.

4. Programmer's View

Having described the design of the tools used to build manageable software, we now discuss the construction of the software libraries. An implementor (one who builds manageable software) must meet certain guidelines in order to develop manageable and reusable software. These guidelines, along with some new approaches to object-oriented design lead to software more in tune with the needs of client code. In turn, programmers implementing clients of such software must be aware of the techniques for managing extending or fine-tuning the software. The next two subsections discuss these strategies from the perspectives of implementors of both the library as well as the client through an example of a socket library.

4.1. Implementor's View

The design of software libraries using the extensible framework is based upon three principles:

- Use of Dual-interfaces [Kiczales, 92b] to create open implementations
- Run-time representation of structure and behavior
- Separate interface and implementation hierarchies

Dual Interfaces represent a strategy based on recent research in language-design [Kiczales, 92a]. Traditionally, black-box abstractions have been used to hide implementation details. Conventional wisdom argues that this approach leads to more reusable and portable code. However, actual interfaces have often revealed implementation details for performance reasons. Implementors have cluttered up interfaces with functions that allow clients control over some aspect of the implementation. Violating encapsulation in this way often ensures that implementations can be fine-tuned by applications. The dual interface approach solves this dichotomy by defining two interfaces. A black-box interface presents the functionality of a software library to clients. An optional second *white-box* interface allows clients to control the implementation through hints and directives. Such an approach gives designers the luxury of basing implementation choices upon actual run-time usage patterns. Examples of choices are setting the message-buffer sizes for a `Socket` class or choosing the hardware link for message transmission.

Run-time manipulation of object behavior and structure requires that behavior and structure be represented in class definitions. As discussed above, behavioral representation in any class is enabled by class-objects accessed through the pointer `meta_obj`, a pointer to the meta-object. Structural representation on the other hand, would be enabled through the ANSI C++ RTTI extensions [Lajoie, 93]. Currently, a prototype library implementation of RTTI allows macro-based run-time type information to be embedded within a class. The declaration of a class needs to include a line of the form:

```
RTTI_SCAFFOLDING_DECL(Name_of_Class)
```

and the definition of the class needs to include a macro of the form:

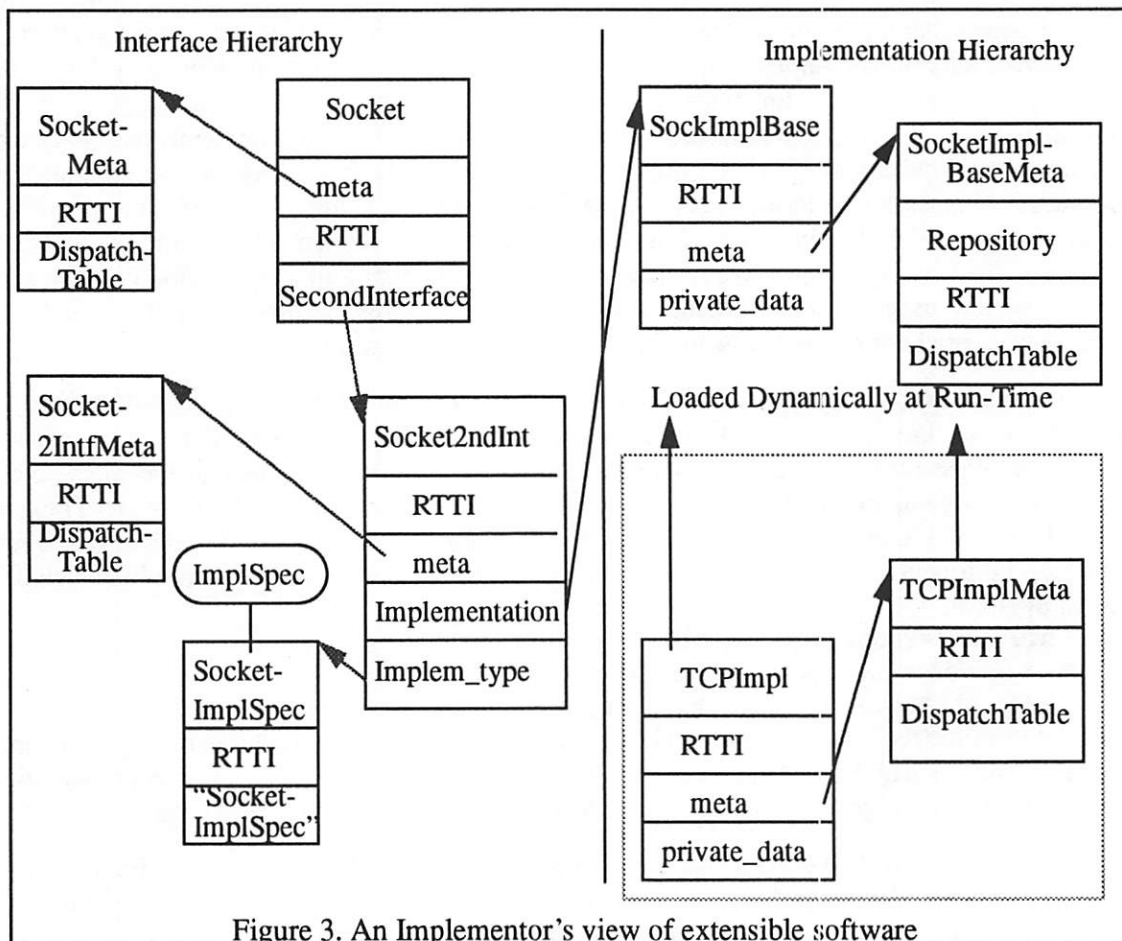
```
RTTI_SCAFFOLDING_IMPL1(Name_of_Class, Name_of_Parent_Class)
```

Then, at run-time the type information pertaining to the class is available through the ANSI specified `get_info()` and the `info()` functions. This structural and type information is an identification mechanism for objects and enables run-time checks in dynamically loaded code.

The fragments of the interface and the implementation hierarchies, as shown in Figure 3, may now be examined. The interface hierarchy contains the primary interface and the dual interface. The primary interface consists of member functions that most UNIX programmers familiar with

sockets would typically use. The dual interface adds the dimension of influence over implementation aspects of sockets, which are typically not under application programmer control. Clients only instantiate the objects that support the main (or primary) interface. The constructors of the main interface class automatically instantiate the dual interface object. The relationship between the main interface and the dual interface, is demonstrated through a hypothetical implementation of sockets, as shown below.

```
class Socket: public SocketBase {
public:
    // constructors and destructors
    // Send a message
    virtual Socket &Send(...);
    // passes on certain calls to the second interface.
    Socket2ndInt *operator->();
    /* member functions for socket functionality*/
    ...
    // macro that generates RTTI scaffolding
    RTTI_SCAFFOLDING_DECL(Socket)
    // pointer to class object that controls
    // the behavioral metacomputation...
    static SocketMeta *meta;
private:
    // Pointer to the optional second interface
    Socket2ndInt *SecondInterface;
};
// specification for the SocketInterface
```



The abstract root class of the implementation hierarchy (`SocketImplBase`) and its class object are the only entities that are visible across the chasm between the two hierarchies. This allows the interaction between the two hierarchies to be limited to the member functions supported by `SocketImplBase` and `SocketImplBaseMeta`. Thus, concrete implementations which inherit from `SocketImplBase` are for the most part limited to the virtual member functions supported by `SocketImplBase`. The only exception to this rule are dynamically added member functions that allow a particular implementation to support additional public member functions. Typically, concrete implementations of the `Socket` interface would inherit from `SocketImplBase` and the class objects of these implementations inherit from `SocketImplBaseMeta`. Concrete implementations may be loaded at run-time or compiled in with the root of the implementation hierarchy.

Finally, in order to support multiple co-existing implementations, interface classes need to be able to identify specific implementations. This support is provided by the `ImplSpec` class hierarchy. `ImplSpec`, as shown below, defines a common protocol for comparison of implementation characteristics. Subclasses of the class `ImplSpec`, may use any characteristic such as the implementation-name to identify implementations, but have to support the comparison protocol defined by `ImplSpec`. Thus, typically every piece of extensible software defines its own subclass of `ImplSpec` in order to distinguish between different implementations.

```
// The following class defines the common comparison protocol, to be supported by all characterizations of implementations. Different extensible software libraries may characterize implementations differently. Some may do it through implementation-name strings, while others may use integer constants.
```

```
class ImplSpec : virtual public CLASS // CLASS is a global base, associated with properties, common to all objects of an extensible library.
{
    public : // some operations have been omitted for brevity
        ImplSpec(const char *); // all characterizations are finally converted into strings, for simplicity.
        virtual ~ImplSpec(); // destructor
        // The comparison protocol follows
        virtual int operator == (ImplSpec &) ;
        virtual int operator == (ImplSpec *) ;
        virtual int operator != (ImplSpec &) ;
        virtual int operator != (ImplSpec *) ;
        ... other comparison operators ...
    private : // mainly responsible for maintaining pointer to a global repository, which catalogs all available implementation characteristics.
        impl_map_t *table; // implementation repository front-end
        ...other private data...
}; // Base class of the Implementation Specification hierarchy
```

4.2. Client's View

A client of extensible software, thus engineered can use the interfaces provided for three purposes:

- Use the direct functionality of the interface e.g.: `Send`
- Use the second interface to control the implementation e.g.: `set_buffer_size`
- Use the meta-functionality to load new implementations of the interface.

The following piece of code demonstrates this for the socket class, discussed above. Initially, a

client programmer allocates a socket object and selects the implementation to be used. For example, in this case the programmer decides to use the UDP protocol for the socket stream. After setting the implementation, the programmer may use the UDP socket as desired. At this point, for some hypothetical and fictitious reason, the programmer decides to use TCP, instead of UDP. Assuming, that the TCP implementation is not pre-loaded, the programmer asks for it to be loaded through the `add_impl` call. Subsequently, the TCP implementation is selected and the TCP socket is ready for use.

```
#include "socket.h"
// Allocate a protocol object - the default implementation is
// used - there may or may not be a default implementation.
Socket *obj = new Socket;

// Actually set the implementation to be used to be udp
(*obj)->set_impl("UDP");
/* The pointer operator is used to get at the second-interface */

// Use the functionality of the socketobject now.
obj->Send(...);
/* Some code here */...

// At this point the client decides to add the tcp implementation
// to the running program. It calls the interface provided by
// the loader class, thru the meta pointer in the Socket class.
obj->meta->add_impl("SocketImplBase", "TCP");

// Now the object may change the implementation type
(*obj)->set_impl("TCP");
// Now, the object can be called regularly, as in..
(*obj)->set_buffer_size(...);
```

Occasionally, the client may want to load an implementation that extends the prescribed interface. Let us assume that the TCP socket is being used. At this point the programmer decides to use the MobileTCP stream, a realization of TCP that supports mobility of connections. Sockets of this stream support an additional member function `migrateconn`. One possible mechanism would be to create an new interface and use it to access the new implementation. However, sometimes recompilation of interface classes is not an option. In such cases, the programmer may dynamically extend the interface of the class, as shown below. The steps to be taken are as follows:

```
#include "socket.h"
#include "MobileTcpSocket.h"
Socket *obj = new Socket;
// Add the new implementation first
obj->meta->add_impl("SocketImplBase", "MobileTCP");
// Now set the implementation to the MobileTcp type
(*obj)->set_impl("MobileTCP");
// Add the interface to the interface hierarchy - here the
// name of the function to be added is "migrateconn"
obj->meta->add_intf("SocketImplBase", "MobileTCP", "migrateconn");
// Some code here to set up connections etc..
....
// Migrate the connection by directly using the loaded function
obj->migrate_conn(...);
```

Thus, clients can directly call the `migrateconn` function, as long as they are using the Mobile TCP implementation. Existing clients do not need to recompile any code. In order to avoid unresolved

externals, the call to `migrateconn` gets converted to a call to `generic_func` at the preprocessing stage. Thus, clients that intend to use this new function must pull in the header files specific to this extended interface, so that the appropriate definitions of `migrate_conn` may be found.

5. Related Work

The need for extensibility in software has been stressed for both operating systems and languages [Kiczales, 92b]. The authors of the Meta-object protocol for CLOS [Kiczales, 91] have been instrumental in discussing open implementations and dual interfaces. Kiczales has also discussed extensibility specifically in the context of object-oriented libraries [Kiczales, 92a]. Lamping has elaborated the purpose and structure of the specialization interface for extensibility through inheritance [Lamping, 93].

Apertos (formerly called Muse) [Yokote, 92] has applied reflection and meta-object protocols to operating systems. The possibility of using frameworks for building reusable software was first discussed by Deutsch [Deutsch, 89]. Brad Cox [Cox, 86] coined the term Software ICs to refer to reusable components, primarily with respect to Objective-C. Choices [Campbell, 93], an object oriented operating system, supports frameworks for dynamic code loading and stepwise refinement. Recently, Open C++ [Chiba, 93] has used translator directives for redirecting method invocations to metaobjects to implement object groups in a distributed system. Aldus has used a custom-built pre-processor to add flexibility to their C++ application framework and to support scripting [Johnson, 93]. In comparison, the significance of this work lies in the structure and extent of extensibility provided in the context of a commercial operating system.

OMG's CORBA [OMG,91] and its implementations such as IBM-SOM [IBM,93] provide some basic facilities for separation of interface from the implementation and interoperability between multi-vendor components. However, our framework goes further by establishing a comprehensive set of metaobjects and providing a reference implementation. It provides facilities for adding to an interfaces, not just for changing an implementation.

The extensibility framework described in this paper may be thought of as a basis for concrete C++/Unix implementation of a set of reusable design patterns. The patterns are comparable to those observed by in ET++ and other similar frameworks [Gamma, 93].

6. Conclusion

The importance of making the management and modification of software cannot be overstressed. The work described in this paper has taken some important steps in making object-oriented software more manageable and reusable. The framework for extensibility and the programming guidelines make the task of implementors considerably easier. The use of dual-interfaces allows for better optimized implementations. Perhaps, the most important contribution is a step towards easing the distribution problems, facing the developers of object-oriented software components.

7. References

- [Banerji, 93] A. Banerji et. al. The Substrate Object Model and Architecture, *Proc. IWOOS '93*, pp. 31-41.
- [Campbell, 93] R. Campbell et. al., Designing and Implementing Choices: An Object-Oriented System in C++, *Communications of the ACM*, 36(9), Sept, 93, pp. 117-126.
- [Chiba, 93] S. Chiba & T. Masuda, Designing an Extensible Distributed Language with a Meta-Level Architecture, *ECOOP '93 - Object-Oriented Programming*, LNCS 707, Springer Verlag, pp. 482-501.
- [Cox, 86] Cox, B.J, *Object-Oriented Programming - An Evolutionary Approach*, Addison Wesley, 1986.

- [Firesmith, 93] D. Firesmith, Frameworks: The Golden Path to Object Nirvana, *Journal of Object-Oriented Programming*, 6(6), Oct. 93, pp. 6-8.
- [Foote, 88] B. Foote & R. Johnson, Designing Reusable Classes, *JOOP*, 1(3),
- [Gamma, 93] E. Gamma et. al., Design Patterns: Abstraction and Reuse of Object-Oriented Designs, *ECOOP '93 - Object-Oriented Programming, LNCS 707*, Springer Verlag, pp. 407-431.
- [Goldberg, 89] A. Goldberg & D. Robson, *Smalltalk-80 The Language*, Addison Wesley, 1989.
- [IBM, 93] SOMObjects Developer Toolkit User's Guide, Version 2.0, IBM.
- [Johnson, 93] R. Johnson & M. Palaniappan, MetaFlex: A Flexible Metaclass Generator, *ECOOP '93 - Object-Oriented Programming, LNCS 707*, Springer Verlag, pp. 502-527.
- [Kiczales, 91] G. Kiczales et. al., *The Art of Metaobject Protocol*, MIT Press.
- [Kiczales, 92a] G. Kiczales & J. Lamping, Issues in the Design and Documentation of Class Libraries, *Proc. OOPSLA '92*, ACM, pp. 435-451.
- [Kiczales, 92b] G. Kiczales, Towards a New Model of Abstraction in the Engineering of Software, *Proc. Workshop on Reflection and Meta-level Architectures, IMSA '92*.
- [Lajoie, 93] H. Lajoie, Standard C++ Update - The New Language Extensions, *C++ Report*, July-Aug. 93, pp. 47-52.
- [Lamping, 93] J. Lamping, Typing the Specialization Interface, *Proc. OOPSLA '93*, ACM, pp. 201-214.
- [OMG, 91] The Common Object Request Broker: Architecture and Specification, OMG Document No. 91.12.1, Rev. 1.1, Object Management Group, Framingham, MA.
- [USL, 92] *C++ Language System*, Unix System Labs.
- [Wirfs-Brock, 90] R. Wirfs-Brock, R. Johnson, A Survey of Current Research in Object-Oriented Design, *Commun. of the ACM*, 33(9), pp. 104-124.
- [Yokote, 92] Y. Yokote, The Apertos Reflective Operating System: The Concept and its Implementation, *Proc. OOPSLA '92*, ACM, pp. 414-434.

Better Widget Design: A Practitioner's Approach

Ed Lycklama, Vice President R&D, KL Group Inc.
260 King St. E, Third Floor,
Toronto, Ontario, Canada M5A 1K3
Ph: (416) 594-1026 x724
Email: eal@klg.com

Abstract: Programming in Motif employs a powerful object-oriented concept based on widgets. This paper discusses the Motif widget model, showing how they are used and briefly discusses how to create a new widget class. Issues involved in designing new high-quality widget classes are discussed, drawing from the author's experience in implementing several commercial Motif widgets.

Motif and Widgets

Every GUI toolkit is made up of components that the programmer uses to build an application. These components are typically objects like push-buttons, sliders and text fields. These objects make it easier to create GUI-based programs without requiring a detailed knowledge of how the underlying windowing system works. Additionally, all programs created with these objects have a consistent look-and-feel.

In the OSF/Motif toolkit, these components are called *widgets*. Widgets are organized into an object-oriented class hierarchy. Every widget class inherits properties from its parent's widget class. A widget class can override or redefine any property it wishes, but in most cases the properties are inherited. In Motif, these properties are called *resources*.

The Motif toolkit is layered on top of *Xt*, the X Intrinsics Toolkit. This toolkit defines a few simple widgets, such as *Core*, but more importantly it provides all the routines and mechanisms necessary to programmatically control widgets. All widget rendering and event management is done through the X11 library, commonly referred to as *Xlib*. This architecture is schematically represented in Figure 1.

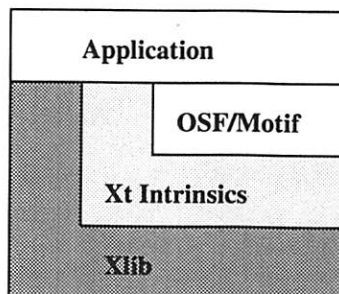


Figure 1— Relationship between Motif, Xt and X11

All Motif widgets are subclassed from the *Core* widget class. This class defines resources such as background color, size and position. Since all widgets inherit these resources, subclasses of *Core* do not need to

re-implement the code for supporting these resources.

There are three classes of widgets: *primitive*, *constraint*, and *shell*. A primitive widget corresponds directly to a simple graphical component, such as a button, slider or text area. Constraint widgets (also referred to as *manager* widgets), are generally not seen; they are used to organize other widgets into a pre-determined pattern. For instance, a row-column widget puts all its “children” into a tabular arrangement, organized into rows and columns. Finally, a shell widget coordinates with the window manager such concerns as geometry (e.g. size and position).

Every widget class is controlled through a common API that allows the programmer to deal with any widget in a consistent fashion. In Xt and Motif, all widgets are primarily controlled with the following six functions:

- `XtCreateWidget` — create a widget
- `XtSetValues` — change resource values
- `XtGetValues` — retrieve current resource values
- `XtManagedChild` — add widget to parents managed list
- `XtRealizeWidget` — create X windows for widget and its children
- `XtDestroyWidget` — destroy a widget and its children

Of course, there are many other functions in the Motif and Xt libraries, but these six (and their variants) are central to every Motif program. The program in Figure 2 illustrates a very basic Motif program which uses all of these functions. It merely displays a single button, which when selected prints out its width and quits the program. This program uses the *Va* variant of the `XtSetValues()` and `XtGetValues()` functions, which allow the programmer to specify a variable list of arguments.

```
1. #include <Xm/Xm.h>
2. #include <Xm/PushButton.h>
3.
4. static Widget toplevel, button;
5.
6. static void
7. quit(Widget w, XtPointer cdata, XtPointer cbs)
8. {
9.     Dimension width;
10.
11.     XtVaGetValues(w, XmNwidth, &width, NULL);
12.     printf("width of button is %d\n", width);
13.     XtDestroyWidget(toplevel);
14.     exit(0);
15. }
16.
17. main(int argc, char **argv)
18. {
19.     toplevel = XtInitialize(argv[0], "Quit", NULL, 0, &argc, argv);
20.
21.     button = XtCreateWidget("quit", xmPushButtonWidgetClass, toplevel, NULL, 0);
22.     XtVaSetValues(button,
23.         XmNlabelString, XmStringCreateSimple("Quit"),
24.         NULL);
25.     XtAddCallback(button, XmNactivateCallback, quit, NULL);
26.
27.     XtManageChild(button);
28.     XtRealizeWidget(toplevel);
29.     XtMainLoop();
30. }
```

Figure 2— Simple Motif Program

Beyond the Motif Widget Set

While Motif provides a good basic widget set, sophisticated applications often find it seriously lacking. For instance, there is no widget in Motif to properly deal with bar charts, pie charts, large tables, or a variety of other objects.

Fortunately, one of the strengths of the Motif toolkit is the ability to subclass existing widgets. This means that any developer can create a new widget class. This new class can be reused in a variety of settings, much the same way that a Motif button can be used in any type of application.

While it is fairly easy to create trivial subclasses of a widget (like a new type of push button that draws its label in a box), in most cases the functionality needed is much different than anything the standard widget set contains. Some examples are graph widgets, surface plot widgets, image viewing widgets, table widgets, and clock widgets.

Although you could write such a widget yourself, there are two alternatives. First, there are many public-domain widgets, and one of these may come close to the functionality you require. This gives you access to the source code, which makes it easy to maintain. Generally, these widgets are little more than someone's solution to an isolated problem, and are rarely supported. Nevertheless, if your needs are simple, and you don't need a commercial-quality product, it is a viable option.

The second alternative is to purchase commercial widgets, which are available from several different vendors. For example, KL Group has three widgets: XRT/graph, XRT/3d and XRT/table. XRT/graph is a graphing widget that displays plots, bar charts, pie charts, stacking bar charts, and filled area charts. XRT/3d is used for displaying 3-D surfaces, contour plots and bar charts. XRT/table encapsulates the concept of a 2-D matrix of editable cells into a widget. Figure 3 shows the position of these widgets in the Motif widget class hierarchy.

Creating a New Widget Class

The process of creating a new widget can be very tedious and time-consuming. However, the task is fairly methodical, and simply requires patience and attention to detail. The process is briefly discussed here — interested readers can consult the references for more information.

Consider a widget which displays itself as a circular light. The state of the light can be on or off. When the light is on, it uses a certain color to distinguish itself from the widget background. The background color is inherited from the Core widget class, so it isn't necessary to do anything with that. Since the widget is quite simple, and does not need to manage other widgets, we will subclass from the XmPrimitive widget (see Figure 3).

We will call this new widget called a XmLightWidget. This widget class has only two new resources: a Boolean resource called XmNlightOn which turns the light on, and an XmNlightColor resource, which specifies the "on" color of the light.

Each of these new resources must be defined in an include file, which we'll call *Light.h*. This include file will be included by any programmer who wishes to use the Light widget. We will also need to define a new resource class for each of these resources.

```
#define XmNlightOn      "lightOn"
#define XmNlightColor   "lightColor"

#define XmClightOn      "LightOn"
#define XmClightColor   "LightColor"
```

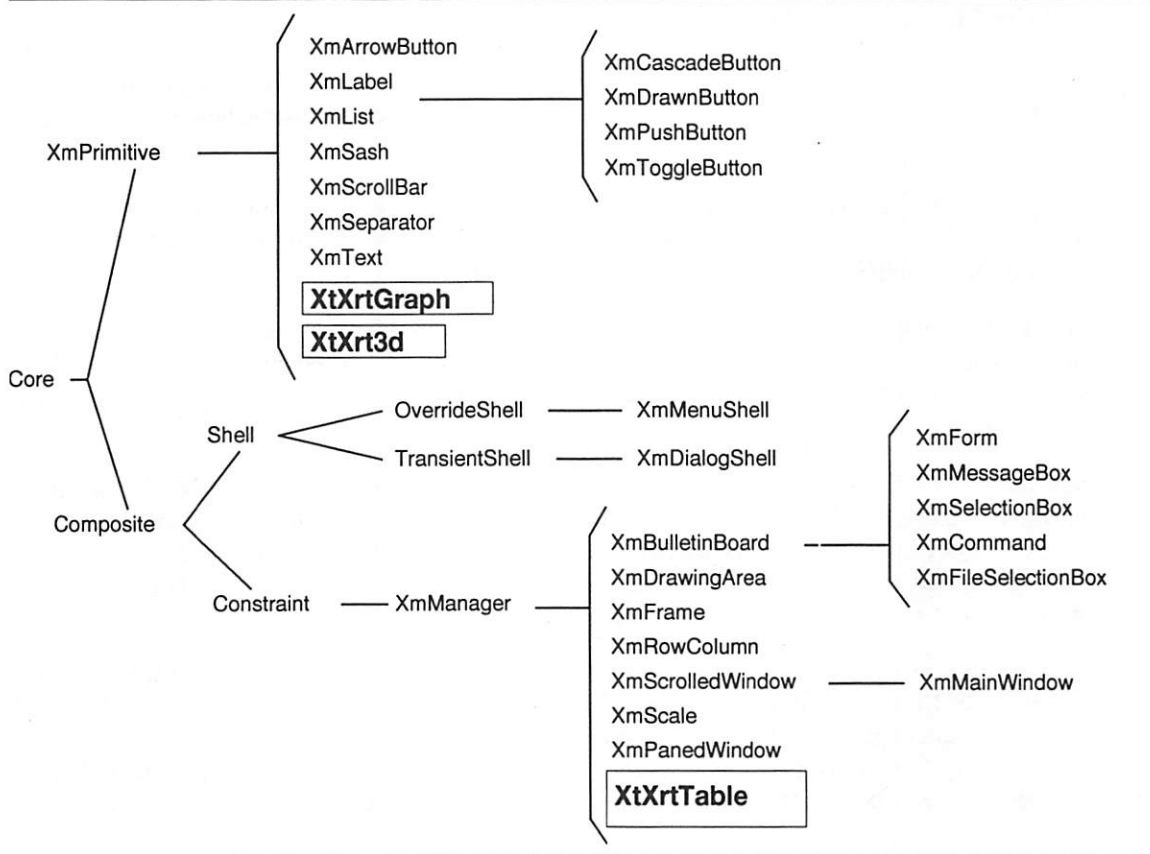


Figure 3— Motif Widgets with KL Group's XRT Widgets

Every widget class must define class and instance structures. Each of these structures must include all of its superclasses structures. The instance structure holds all the values that define a particular instance of the widget, in this case the state and color. For most widgets, the class structure defines nothing new.

The widget class and instance structures are defined in the file *LightP.h*. The widget instance and class records are defined as follows:

```

typedef struct _XmLightPart
{
    Boolean    is_on;
    Pixel      color;
} XmLightPart;

typedef struct _XmLightRec
{
    CorePart      core;
    XmPrimitivePart  primitive;
    LightPart     light;
} XmLightRec;
  
```

```

typedef struct _XmLightClassPart
{
    caddr_t          extension;
} XmLightClassPart;

typedef struct _XmLightClassRec
{
    CoreClassPart      core_class;
    XmPrimitiveClassPart primitive_class;
    XmLightClassPart    light_class;
} XmLightClassRec;

```

Notice how the subclassing is explicitly defined by the inclusion of each of the superclass parts up the chain to the Core widget class (see Figure 3). This is all tied together in the widget implementation file, *Light.c*. This initializes the widget class structure, as shown in Figure 4. Each line that is highlighted distinguishes the XmLight widget class from other widget classes.

Line 14 specifies the list of resources for this widget. This is an array of structures initialized as follows:

```

static XtResource resources[] = {
    { XmNlightOn, XmCLightOn, XmRBoolean, sizeof(Boolean),
      XmPartOffset(XmLightRec, light.is_on),
      XmRImmediate, (XtPointer) False,
    },
    { XmNlightColor, XmCLightColor, XmRPixel, sizeof(Pixel),
      XmPartOffset(XmLightRec, light.color),
      XmRString, (XtPointer) "green",
    }
};

```

Each of these structures specifies the resource name, class, resource type, number of bytes required to represent the value, the offset into the resource structure, and a default value for the resource. Note the state of the light is specified in place (XmRImmediate), whereas the color of the light is specified as a string. In this case, a resource converter is called at run-time to convert the color name into a pixel value.

Several widget methods must be written to implement basic widget behavior. These have been highlighted in Figure 4, and are summarized below:

ClassInitialize	This is called the first time an XmLight widget is created. It must initialize the widget class structure, and install any resource converters that this widget class requires. All the part offsets are also calculated, using <i>XmResolvePartOffsets()</i> .
Initialize	This is called as a result of the programmer calling <i>XtCreateWidget()</i> for the XmLight widget class. Any resource that the programmer passed in as a parameter, as well as any pertinent resources from resource files, are merged together and applied to the widget. Note that the resource values are stored in the widget structure by Xt, but it is the widget writer's job to check each of these values for range errors. Also, any value which is a pointer should be duplicated so that the widget has its own copy of the resource value.
Realize	This is called immediately after an X Window is created for this widget. Any widget information which relies on server information must be completed at this time (colors, fonts, etc.).
Destroy	This is called when the widget is being destroyed. The widget must free all resources it has requested, whether they be allocated memory or server resources.

```

1. XmLightClassRec xmLightClassRec = {
2.     [ /* Core */
3.         (WidgetClass) &xmPrimitiveClassRec, /* superclass */
4.         "XmLight", /* class_name */
5.         sizeof(XmLightPart), /* widget_size */
6.         ClassInitialize, /* class_initialize */
7.         NULL, /* class_part_initialize */
8.         FALSE, /* class_inited */
9.         (XtInitProc) Initialize, /* initialize */
10.        NULL, /* initialize_hook */
11.        Realize, /* realize */
12.        NULL, /* actions */
13.        0, /* num_actions */
14.        resources, /* resources */
15.        XtNumber(resources), /* num_resources */
16.        NULLQUARK, /* xrm_class */
17.        TRUE, /* compress_motion */
18.        XtExposeCompressMultiple, /* compress_exposure */
19.        TRUE, /* compress_enterleave */
20.        FALSE, /* visible_interest */
21.        (XtWidgetProc) Destroy, /* destroy */
22.        (XtWidgetProc) Resize, /* resize */
23.        (XtExposeProc) Redisplay, /* expose */
24.        (XtSetValuesFunc) SetValues, /* set_values */
25.        NULL, /* set_values_hook */
26.        XtInheritSetValuesAlmost, /* set_values_almost */
27.        NULL, /* get_values_hook */
28.        NULL, /* accept_focus */
29.        XtVersionDontCheck, /* version */
30.        NULL, /* callback_private */
31.        NULL, /* tm_table */
32.        NULL, /* query_geometry */
33.        NULL, /* display_accelerator */
34.        NULL, /* extension */
35.    ],
36.    [ /* XmPrimitive */
37.        _XtInherit, /* border_highlight */
38.        _XtInherit, /* border_unhighlight */
39.        XtInheritTranslations, /* translations */
40.        NULL, /* arm_and_activate */
41.        NULL, /* syn_resources */
42.        0, /* num_syn_resources */
43.        NULL, /* extension */
44.    ],
45.    [ /* XmLight */
46.        NULL, /* extension */
47.    ]
48. };

```

Figure 4— XmLight WidgetClass Definition

Resize	This called whenever the widget has changed size due to a user request. The widget should recalculate layout, but not repaint itself.
Redisplay	This is called whenever the widget window receives an expose event. The region of the widget which is affected by the event is also passed, so the widget may use this to only repaint certain portions of the widget.

SetValues

This is called whenever the programmer calls *XtSetValues()* on this widget. The widget should examine each resource that has changed value. It is up to the widget programmer to make sure that the passed value is valid. If the resource points to a block of memory, any previous copy that the widget had made must be freed. Also, any side-effects that a resource may have should be applied in this method. This method is usually the most complicated and time-consuming to write.

Widget Design Considerations

The remainder of this paper will discuss some of the issues involved in designing new widgets. The discussion will be broad enough to be useful to those evaluating public-domain or commercial widgets, as well as developers writing new widgets.

The issues are grouped into three broad classifications: basic widget design, managing resources, and advanced issues.

Basic Widget Design

There are many issues inherent to the design of any new piece of software that apply as well to designing new widgets. As with any other piece of software, giving ample time to the design process can save countless hours spent in coding and testing. Here are some basic issues:

Keep it Simple (Stupid!)

A good widget should do simple things simply. Any parts that are complex, or are considered "advanced" usage, shouldn't concern the programmer who is not interested in them. This allows a programmer to get started quickly with a new widget, only learning new features as it becomes necessary.

Conforms to OSF/Motif Standards

A good widget should look like it belongs in the toolkit for which it is being considered. Consult the OSF/Motif style guide to make sure the design of your widget, and how the user interacts with it, will complement the OSF/Motif widget set. The end-user should think that your new widget is just part of the standard widget set. For example, all new widgets should be able to use the mouse-less keyboard traversal mechanisms employed by Motif. Also, any new widget should use "shadows" similar to Motif around their border.

Side Effects

Resources should have as few side-effects as possible. Where necessary, apply them in a consistent and easy-to-understand fashion. For instance, changing the data for a graph should have obvious side-effects on the default axis origin; the converse is obviously not true.

Reusable

Regardless of the task for which you are considering a particular widget, make sure that the widget will be usable in a variety of other tasks. Ultimately the widget should be able to be used in many diverse applications within your organization, so ask yourself how well it will adapt to different situations.

Quick and Efficient

Resource value changes should take effect quickly and efficiently. Very large Intrinsics based widgets may have trouble with this because the internal widget mechanisms imposed by Xt are tailored to small widgets with simple resource types. Widgets which have many resources with complex resource types cannot use the method that most Xt programming books suggest.

This is most evident in the widget method `SetValues`. This function is passed a copy of the old widget (before the values were applied), and the new widget. If the widget developer is interested in exactly which values changed (and they are), the suggested procedure is to compare each of the resource values in the old and new widget. This is acceptable if there are few resources, but it quickly becomes inefficient as the number of resources grow.

A better way is to look at the fourth argument passed in, which is the actual argument list of the resources the programmer wants to change. Each of these resources can be processed and handled in a much more efficient manner than comparing all the old and new values of each resource.

Robust

Resource values should be checked for reasonable values. The design of Xt makes this difficult, because resource values that are set at widget creation go through a different path than when set after widget creation (`Initialize` vs. `SetValues` method). This forces the widget programmer to check the values in both methods. Since any widget resource can also be set in a resource file, it is necessary to check all resource values in both locations.

Library Dependence

If you are looking to purchase a widget library, make sure there are no libraries (particularly proprietary ones) other than the Xm, Xt, X11 libraries that the widget depends on. If a widget depends on any other libraries, particularly if the library needs to be purchased from the vendor, then the widget is likely just a front-end to a procedurally-based library, and may not have a good support for resources.

Managing Resources

The fundamental aspect of any Motif widget is its dependence on resources to drive it. Resources (such as background color, font, label string, etc.) can be set by the programmer, or can be set by the user in resource files. If you opt for a procedural interface to the widget, you take away the ability of the user to customize the appearance of the widget.

For instance, when we designed the XRT/table widget, we needed to come up with a resource-based solution to the problem of specifying resources like colors and fonts on a per-table, per-row, per-column and per-cell basis. The solution we devised was to implement a pair of resource types: one based on a *context*, and one based on a *series*. A context is a resource that specifies either a cell, row, column, or entire table, while a series gives a list of context-defined resources.

For example, to specify the background color of cell (2, 3) as red, and column 4 as blue, there are two different methods. Using contexts, one would write the following:

```

XtVaSetValues(table,
    XmNxrtTblContext, XrtTblSetContext(2, 3),
    XmNxrtTblBackgroundContext, "red",
    NULL);
XtVaSetValues(table,
    XmNxrtTblContext, XrtTblSetContext(XRTTBL_ALL, 4),
    XmNxrtTblBackgroundContext, "blue",
    NULL);

```

To set this using series, the most convenient way is to specify the resource in a resource file:

```
*.table.xrtTblBackgroundSeries: (2 3 red) (ALL 4 blue)
```

Both series and context resources resolve to the same internal structure, so there is no difference. One method is more convenient for the programmer, and the other more convenient for end-users. Since it is all resource based, access to all the visual controls of the table is accessible through every standard access mechanism, including integration through standard GUI builders such as UIM/X and Builder Xcessory.

When you're designing a new widget, every resource should have a reasonable default value. This is easy enough when there is a fixed default which will do (for instance a default shadow thickness of 2 pixels), but sometimes the default value should be dependent on some other resource. For instance, in a graph plot, what value is reasonable as a default for the maximum Y axis value? Clearly, this default is dependent on the data; there is no single value which is a reasonable default.

If this is the case, then the method of specifying that the widget should dynamically calculate the axis maximum is problematic. If you assign a special value to this resource to mean "use the data maximum", then if the programmer wishes to determine what the axis maximum is, using *XtGetValues()* is not going to return the right answer.

The way we solved this with XRT/graph is to use a pair of resources. For example, *XtNxrtYMax* and *XtNxrtYMaxUseDefault*. The second resource is a Boolean resource which says whether or not XRT/graph should calculate the default value for the *XtNxrtYMax* resource. When False, the actual value in the *XtNxrtYMax* is used; otherwise, the widget calculates the value for *XtNxrtYMax*.

XRT/graph uses this type of resource pair for each axis minimum, maximum, origin, number, tick and grid increment. Since the default value of each Boolean is True, the programmer need never assign any values to the floating-point resources. As a result, the programmer is able to produce a graph in a very short time.

Every resource defined by the widget must be assigned a resource type. Often, one of the resource types defined by Motif can be used (Boolean, String, integer, etc.). However, any complex widget will need to define its own resource types. These are often enumerated types, but sometimes can include structure or array pointers.

One of the more common problems is the lack of a Float type in Motif. This means that there is no way to specify a resource of type *float* or *double* in Motif. All of arguments passed to *XtSetValues()* must be of type *XtArgVal* (usually a *caddr_t*, big enough to fit any pointer). A particular problem with the C programming language is that any parameter of type *float* is automatically promoted to type *double*. Hence, the following code will not work:

```

XtVaSetValues(graph,
    XtNxrtYMax,      22.0,
    NULL);

```

There are two common solutions to this problem. One is to pass in the address of the floating-point param-

ter, the second to coerce the value into something of size `XtArgVal`. Neither are very pretty. For XRT/graph we hid the implementation details inside the function `XrtFloatToArgVal()`. The following code fragment will work in XRT/graph:

```
XtVaSetValues(graph,
               XtnXrtYMax,      XrtFloatToArgVal(22.0),
               NULL);
```

This function uses a union containing both a `float` and `XtArgVal` member. The passed value is assigned to the `float` member, while the `XtArgVal` member is returned. This works because on most architectures, `sizeof(float) == sizeof(XtArgVal)`. Obviously, a different solution is required on those architectures where this is not the case.

Regardless of the type of resources defined in your widget, you will probably need to write some resource converters. These converters are used to convert between strings (from resource files) into the proper resource type expected by the widget. Motif has built-in resource converters for all its resource types, so you'll only need to write new ones for the resource types you have defined.

A common issue with resource converters is resource caching. This is the mechanism by which Xt will cache the results of previous conversions. However, if you are writing a custom widget, chances are you will not be converting the same resource value repeatedly. Also, if you do cache the results, you will probably be creating a memory leak in the caching mechanism, whose memory you are unable to access. Hence, for most widgets it does not make sense to cache custom resource conversions.

Advanced Issues

There are many other issues that are part of designing a widget. Most of the issues discussed in this section are considered "advanced" from the widget writer's point of view. However, to an end-user, issues such as printing, user interaction and professional documentation are considered essential.

Printing Capability

Widgets that convey useful information to the user should understand how to print themselves. This applies to widgets such as gauges, graphs, and tables. Although X defines no standard API for printing, today's modern applications demand it. Printing support is an essential part of a graphical object.

Memory Leaks

No widget should leak memory, nor cause the server to leak memory. Applications that are put into production stay running for days, if not weeks or months, at a time. A widget which is continually updating its display (e.g. graph, dial, gauge, clock widgets) must not allocate any memory that it does not also free. Failure to do so adversely affects the performance of the machine it is running on.

Also, the widget should not leak memory when it is continually destroyed and re-created. Unfortunately, current releases of Motif and Xt will leak memory during a create/destroy cycle, so if you really care about zero memory growth, try to avoid creating and destroying a lot of widgets.

User Interaction

A widget should not just provide display capabilities, but should also be able to respond to user events. Most widgets that are part of the standard toolkit provide both display and user interaction — why should a custom widget be any different? For example, a graph widget should provide information about pointer selections on the widget, and a 3-D surface widget should provide rotation capabilities.

Dynamic Offset Calculations

This is a very poorly documented feature of Motif, but it is important if the widget is to be compatible with future releases of Motif. This is evident from the way subclassing is done in Motif. Since every class directly includes the structures which define its superclasses, the compiled offsets will be incorrect should the new widget should it be linked against a version of Motif in which any of those included structures changes.

Most (if not all) books suggest using the `XtOffsetOf` macro for your resource list; however, you should use the `XmPartOffset` macro instead, and then call `XmResolvePartOffsets` to calculate the actual field offset. Unfortunately, this function assumes that each of the structures requires a four-byte alignment. If you use any `double`'s in your widget definition, this will obviously be false, and the offsets will be calculated incorrectly. In this case, you will have to roll your own.

Documentation

For a complex widget, the documentation is a crucial part of the product. The documentation must get the programmer started quickly using the widget. It should clearly describe the basic concepts of the widget, direct you towards correct usage of the widget, list the supported resources, and guide you through common programming tasks.

Programming Examples

There is nothing as illuminating as good programming examples. Clearly documented code should take you through some simple and not-so-simple widget usage. Used in conjunction with a good set of documentation, programming examples are invaluable in the accelerating the learning curve.

IDT and UIMS Integration

The task of designing a user interface can be tedious without the aid of a good Interface Development Tool (IDT) or User Interface Management System (UIMS). Most of these tools now support GUI development using third-party widgets, but for any non-trivial widget the integration takes some effort. In particular, any widget which defines its own resource types will require specialized resource editors to be built into the tool.

Internationalization Support

Although not important to all developers, for some it is imperative. If the widget only works in certain locales (as defined by ANSI C), then you will not be able to sell a product using this widget outside those locales. The support required depends on the widget in question, and could range from correct formatting of numeric values (graph widgets), proper time representation (graph, clock and calendar), to support for two-byte fonts (e.g Kanji).

Summary

Widgets form the basis for all GUI work in the X Window System today. Their quality, diversity and availability directly affects application developer productivity and end-user satisfaction. This paper has examined some of the issues involved in developing high-quality widgets, and how they affect application developers and end-users.

Good quality widgets are available today, but not all widgets are good quality. A complex widget requires careful design and detailed understanding of the Motif, Xt and Xlib components they depend on. As this market matures, we should see a greater number of quality components for a wide range of vertical and horizontal markets.

References

1. Adrian Nye and Tim O'Reilly. *X Toolkit Intrinsic Programming Manual*. O'Reilly & Associates.
2. Paul Asente and Ralph Swick. *X Window System Toolkit*, Digital Press.
3. Dan Heller. *Motif Programming Manual*, O'Reilly & Associates.
4. *OSF/Motif Programmer's Guide*, Release 1.2, Prentice Hall.
5. *OSF/Motif Programmer's Reference*, Release 1.2, Prentice Hall.
6. *OSF/Motif Style Guide*, Release 1.2, Prentice Hall.
7. R.W. Schiefler and Jim Gettys. *X Window System*, Digital Press.
8. Alistair Gourlay. *Writing Motif Widgets: A Pragmatic Approach*, in The X Resource, Issue Six, O'Reilly & Associates.

Implementing A Generalized Drag-and-Drop in X

Cui-Qing Yang and Shrinand Desai

Department of Computer Science
University of North Texas
Denton, TX 76203-3886
cqyang@cs.unt.edu

Abstract

Drag-and-drop operation is a well-recognized and widely applied mechanism in the development of graphical user interfaces (GUIs). Most of current approaches in design and implementation of such a operation are limited to some specific objects (such as icons or drag-and-drop widgets), and are usually complex, system-dependent, and time-consuming. This paper proposes a generalized semantic and protocol for drag-and-drop that makes the draggable and droppable as intrinsic properties of any GUI objects (just like the colors and other appearance of an object) and allows each object to customize its own semantic of drag-and-drop. Therefore, the drag-and-drop feature is integrated as an inherent part of a GUI window system and all details and complexities of its implementation can be hidden from end users. The framework of generalized drag-and-drop could be applied to any GUI system. A prototype implementation in the Xt Athena widget set of X11R5 on a Unix system is reported in the paper.

1 Introduction

As the use of computers continues to grow, so does the need for good and effective graphical user interfaces (GUI). Well-designed GUIs can make the learning process easier and aid in the use and understanding of applications. With support of different window systems on many machines, such as the Macintosh, the MicroSoft Windows, the IBM OS/2 Presentation Manager, and the X Window System, applications of GUI are now based on the manipulation of various GUI items, e.g., windows, menus, icons, scroll bars and buttons. Among them, one of the most attractive operations in a window system is drag-and-drop, with which a user can use the mouse to pick up a source item, drag it elsewhere on the screen, and drop it on some destination site so that a predetermined operation can take place. In this way, drag-and-drop emulates interactions between different types of objects in a user application. For example, a data file (representing a source) can be dragged with the help of a mouse, and dropped on a destination site (probably an icon representing printer) in order to print the file. The advantage of this is that a general user does not have to be concerned with the nitty-gritty details of various printer command syntax. Thus, GUIs, with the aid of devices like a mouse and proper usage of menus and drag-and-drop operations, become more intuitive and more friendly for end users.

Due to their ease of use and inherently intuitive nature (the way we think), drag-and-drop operations have gained popularity in building friendly user interfaces. However, this

advantage of drag-and-drop comes with its own stock of problems. Firstly, it is rather involved to integrate such operations of drag-and-drop in an application program, since this requires programmers to handle low-level details that a user should usually not be expected to know. These details prove cumbersome even for a skilled programmer. Secondly, as drag-and-drop demands a smooth communication among a wide variety of non-similar applications, an universal protocol needs to be followed among the applications participating in these operations. At present, many applications are available that provide consistent mechanisms for icon drag-and-drop, e.g., the HP VUE, the IXI X.desktop [2], and the OpenWindows of SunSoft [5]. Nevertheless, their implementations are more or less ad hoc, complex and not-reusable. There lacks a general framework in supporting the operations of drag-and-drop in an arbitrary client application of most window systems.

In this paper, we present a new framework of generalized drag-and-drop for GUI window systems. The scheme is based on the object-oriented programming paradigm and provides a generalized semantic and protocol for drag-and-drop operations in a GUI system. Instead of treating drag-and-drop as specific features of specific objects, we consider *draggable* and *droppable* as two of the basic properties of any GUI objects, just like other properties such as the colors and sizes, of an object. Furthermore, mechanisms are provided for individual objects to customize these properties with *True* or *False* and to specify the semantics of what is to be dragged and what action is to be taken when a drop operation is performed within an object. With such a generalized form, the feature of drag-and-drop is integrated as an inherent part of a window system and all details and complexities of its implementation can be hidden from end users. Some of the major advantages of this method over existing techniques in design and implementation of drag-and-drop operations are as follows:

- The treatment of drag-and-drop operation as the basic properties of a GUI object follows the same spirit of object-oriented programming paradigm for GUI applications, therefore, preserves the benefits of modularity, reusability, and maintainability.
- The properties of “draggable”, “droppable” and their associated mechanisms for customization provides high flexibility in implementing operations of drag-and-drop, encapsulates low-level details from end programmers, hence, greatly eases user efforts in design and implementation of these operations.
- The high-level abstraction of “draggable” and “droppable” properties of GUI objects gives a uniform framework to handle drag-and-drop operations in various GUI window system and enhances the portability of GUI code between them.

We believe that the concept of generalized drag-and-drop should be applicable to many different GUI systems. A prototype implementation of the proposed scheme has been completed in X11 R5 using Xt Intrinsics and Athena widget set [15]. Applications based on properties of new widget classes demonstrate the ease with which customized drag-and-drop operations are realized in GUI applications.

The paper is organized in the following manner. After a brief review of the related work in Section 2, Section 3 address the main issues in defining the generalized drag-and-drop. The implementation details of the widget hierarchy, the new properties of widget classes, and the protocols for inter-client communications are discussed in Section 4. Section 5 presents some testing results, and finally, Section 6 concludes the paper.

2 Related Work

The idea of using icons to represent objects in a programming environment, and the implementation of icon drag-and-drop started since the early days in the development of GUIs. The pioneering work of the Xerox Star system in the early 80's [13, 7] introduced the concept of "Desktop", which resembles the top of an office desk representing the working environment of users of computer systems. All objects, such as documents, folders, file drawers, in-baskets, and out-baskets, are displayed as small pictures called "icons". The icons are visible, concrete embodiments of the corresponding physical objects. Various operations are supported for the icons, including drag-and-drop. For instance, a document icon can be "dragged" (moved) around the Desktop and "dropped" onto the out-basket icon, which will mail the content of the document to its destination. The design principles of the Star user interface greatly simplify the human-machine relationship, making computer systems seem familiar and friendly to most users, and open a new direction for computer system design.

The success of the Apple Macintosh system and its user interface is a historical evolution in the innovative concept of desktop and icon manipulation originating from the Star system. It is the Macintosh system that fully explores the potentials of the desktop metaphor and puts it into massive implementation. As the focus of the Macintosh user interface, the Macintosh Finder [3] supports a desktop environment, which manages all the resources and objects an application programmer needs. All objects are represented by icons, including the documents, folders, applications and disks. Operations on the objects, such as open, copy, move, discard, rename, and lock, can be performed via the manipulation of their icons. Among them, the icon drag-and-drop is one of the best features.

Both Star DeskTop and Macintosh Finder support the usage of icon drag-and-drop for user applications. Mechanisms are available for users to specify customized properties for the appearance and behavior of icons representing their applications. However, the implementation of icon drag-and-drop are proprietary of the companies, and are deeply embedded in the systems. No general framework is provided for the adoption of icon drag-and-drop operation for arbitrary applications in these systems.

With all of its power and features for GUI development, the X Window System offers a perfect vehicle for the implementation of GUI applications. There are X-based desktops and applications available on many different systems that provide operations of icon drag-and-drop. Some of them are HP VUE, IXI's X.desktop [2, 6], and the OpenWindows of the SunSoft [10]. The OpenWindows is a development tool designed for building graphical user interface for user applications [16]. It is based on the X Window System, and supports programs in three different Toolkits of Sun Microsystems – the XView Toolkit, the NeWS Toolkit, and the OPEN LOOK Toolkit. With the help of glyphs palette, panes, and menus in the OpenWindows, users can easily select various UI elements (User Interface elements) and assemble them together into an interface for their application. This gives programmers the freedom to create and test user interfaces without writing any code.

The OpenWindows supports nice features for implementing drag-and-drop in general user applications. For that purpose, *drag* and *drop targets* are defined as regions in control areas where drag and drop operations can be performed. Users can simply choose the instances of these targets in their targeted interfaces. Edit windows are provided in the OpenWindows that allow users to specify a variety of properties for the selected drag and

drop targets. These properties, such as the label, the location (x, y coordinates), the color, and the cursor, define the appearance and behavior of the drag and drop targets. In this way, users can easily customize drag-and-drop operations in any of their individual applications. This makes the OpenWindows different from other desktops like the Macintosh Finder and the Star system.

3 A Generalized Drag-and-Drop

In many GUI systems, drag-and-drop is implemented as specific properties of only limited GUI objects, such as icons or windows. Also, the semantics and implementations of drag-and-drop have always been system specific. Moreover, many GUI systems support the concepts of "clipboard" and "cut-and-paste" that can be used to move around some text or graphics contents from one application to other applications. Both of these features have similar semantics and functionalities as the drag-and-drop operation, however they are usually handled differently from the implementation of drag-and-drop. There lacks a uniform model in providing the semantics and functionalities of drag-and-drop in different window systems. This leads to the difficulty and complexity in the design and implementation of drag-and-drop operation in general GUI applications. The basic considerations of our framework are as follows:

- *Drag-and-drop as a basic property*¹ - Drag-and-drop is considered as a basic property of any GUI (widget) classes and their objects just like other properties as colors and sizes. For this purpose, two Boolean flags: *draggable* and *droppable* are defined in every widget class, and each object instance of a class can customize these properties with *True* or *False*, which indicates whether or not the object supports the operation of drag-and-drop.
- *A generalized semantic of drag-and-drop* - With our approach, the drag-and-drop concept becomes very generalized and more flexible. It lets applications customize the drop operation as well as the drag operation in any specific widget instance (object). The semantic provides a general framework for dragging and dropping an object from a drag site to drop site, but it does not restrict the functionality or operations. These are completely left to the application programmer in customizing the operations at each specific object. For example, in one instance, the user may decide to specify the operation of dragging a file name in a directory object and dropping it on another directory object as to move a file from one directory to another, and on another instance, the user can decide to customize the same drag and drop site so that only the color of one directory window is moved (copied) to another directory window. In order to support such a generalized drag-and-drop, two callbacks (procedure properties) are added to each widget class: *dragcallback* and *dropcallback*, with which applications can customize operations of dragging and dropping in each individual widget object.
- *A general protocol for drag-and-drop* - One important component of the generalized drag-and-drop is the protocol for communications between the source (dragging) and

¹Properties of GUI objects are often referred as *resources* in many GUI systems. In this paper, we use both term interchangeable unless it is specifically noted.

destination (dropping) objects. Generally, the protocol requires the sharing of certain information between source and destination objects, including the identification of the dragging object, the draggability of the object, the source data associated with the dragging object (that is generated from the dragging callback), the location and identification of the dropping object, its droppability, and the operations (callback) of the dropping object.

There can be different approaches for protocols sharing information between source and destination objects. A straightforward way is to have source and destination communicate directly using messages (such as the *ClientMessage* in X) [6]. That is, during the course of dragging a source object, the source can first initiate a message to the destination with the information of its identification, and then the destination will acknowledge a message back to the source with some additional information. More messages can be exchanged between source and destination if they are needed. Another way to arrange communication between source and destination is not to have them exchange messages directly. Instead, a third party, such as the window manager in X, can be involved as an intermediary between the source and destination for exchanging information. There are some more ways to arrange the drag-and-drop protocol in which the messages between the source and destination can be reduced and even eliminated. One of such a protocol is to have a shared data structure (database) between the source and destination. Each destination is required to pre-register its information in the database (the identification, the location, and the callback), and the source can search for the related information after it is dragged [6].

The framework of the generalized drag-and-drop imposes no constraints to the specifics of the protocol as far as there is such a protocol to support communications between the source and destination objects. It is up to the implementors for the decision of general strategy and detailed steps in a protocol. Certainly, a standardized protocol will help the portability of the generalized framework, which is beyond the scope of this paper. A specific protocol based on a shared database of dynamic registration was chosen in our testing implementation presented in the paper.

Also, a complete drag-and-drop protocol specifies all details for the interaction and feedback between the source and destination objects, such as the change of placement and visibility of the source during dragging, and the change of visibility of destination at dropping. For simplicity, our discussion ignores some of those details and only concentrates on the main interactions (the data exchange and callback invocation) of the protocol.

The framework of our generalized drag-and-drop does not depend on any specific GUI system and the hardware platforms. As our model is based on the object-oriented approach and the abstraction of GUI classes, it can be integrated to the most of current GUI window system without much difficulty and provide a common ground in the design of drag-and-drop operations.

4 Prototype Implementation

In this section we present a prototype implementation of a generalized drag-and-drop. For simplicity the prototype was limited only to certain necessary resources. The following subsections go through the widget hierarchy, resources and protocol typical to this prototype.

4.1 Widget Class Hierarchy

We choose to use Xt Intrinsics and the Athena widget set (Xaw) [15, 5] as the testing ground for our prototype implementation. Xaw has a fairly complete sets of widget classes. More importantly, it comes with a full source code, which makes it easy to alter it and test our implementation.

Generally, Xt Intrinsics provides basic mechanism to create new widget classes. Xt Intrinsics defines the Core class as the basis of all widgets in its toolkit. Thus, the class record of the Core class embodies information common to all widgets [1]. Certainly, we could have implemented our prototype in the Core widget. However, as our first effort, for simplicity we limited our hierarchy to the Simple widget in Xaw, a direct subclass of the Core class (in this way, we avoided to alter any code in Xt Intrinsics). Simple widget class is a very general widget class that has a rich set of children classes such as List, Text, and Label. All of its children inherit properties like window and cursor from the Simple widget class. Figure 1 shows the class hierarchy of the Simple widget class.

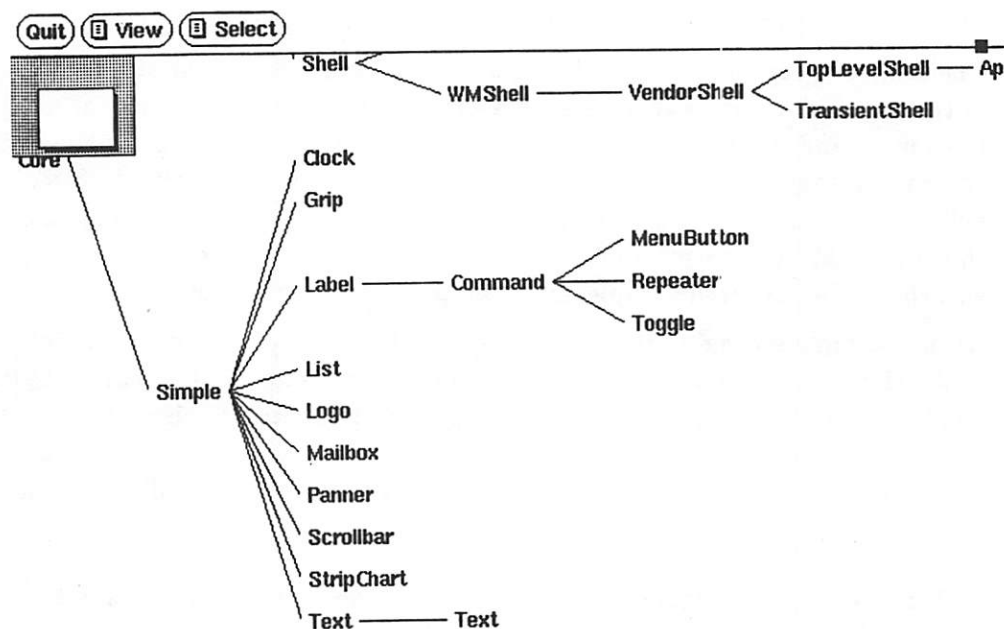


Figure 1: Simple Widget Class Hierarchy in Xaw

Besides, in order to further test our implementation, we also made a whole new widget class called Icon widget class as a child of Simple widget class. The Icon widget inherits properties of drag-and-drop from Simple widget class, and can be used for testing drag-and-drop operations of icons, which is popular in many GUI systems.

4.2 New Properties (Resources)

Since the Simple widget class is directly derived from the Core widget class, it automatically inherits all the properties defined in the Core. Additionally, we need to define new properties that are specific to the Simple widget class for the sake of drag-and-drop operations.

Four public resources are added to Simple widget class: 1) draggable, 2) droppable, 3) dragCallback, and 4) dropCallback. (Figure 2 shows some selected resources used by Simple widget class). These resources can be set by an application to inform a widget if it is draggable or droppable or both, and to define the functions associated with dragging and dropping operations. Since the dragging and dropping feature is treated as resources, it provides all the advantages and flexibility of resource settings.

Name	ClassRep	Type	Default Value
background	Background	Pixel	XtDefaultBackground
border	BorderColor	Pixel	XtDefaultForeground
borderWidth	BorderWidth	Dimension	1
cursor	Cursor	Cursor	None
cursorName	Cursor	String	NULL
destroyCallback	Callback	Pointer	NULL
height	Height	Dimension	0
insensitiveBorder	Insensitive	Pixmap	Gray
pointerColor	Foreground	Pixel	XtDefaultForeground
sensitive	Sensitive	Boolean	True
width	Width	Dimension	0
x	Position	Position	0
y	Position	Position	0
droppable*	Droppable	Boolean	False
draggable*	Draggable	Boolean	False
dragCallback*	Callback	Pointer	NULL
dropCallback*	Callback	Pointer	NULL

Figure 2: Resources of Simple Widget Class
(Note: * indicates the new resources added)

Some private variables are also added to Simple widget which are not visible to user applications. These resources (e.g. source, destination, source and destination type and size) are used internally by Simple as well as the subclasses of Simple widget. These resources are available in a read-only form to an application when the user sets callback routines for draggable and droppable widgets. Some information available to a user through a callback are 1) source (window) id, 2) destination (window) id, 3) source data pointer, and 4) mouse position (x and y coordinates) where the source was dropped, which can be useful when a user wants to pop up a selection box at the drop site. The source type can be either text or widget ID. At this point the drag and drop feature is truly a general feature. In future, when different types of data can be dragged and dropped, more fields of type information need to be added in the class record.

4.3 The Protocol

In order for any drag-and-drop operation to take place successfully every source object must have information about all the potential drop sites or destination objects. Moreover, potential dropsites may be in different applications that have no knowledge of one another therefore there must be some kind of protocol support for cooperation between source and destination objects. That is, information must be available to every source from each destination and must be dynamically updated from time to time. Based on this information, when a source is dragged, it can be determined whether the pointer is in a dropsite, whether a drop would succeed, and what the resulting operation would be.

In our implementation, this form of information-sharing is achieved by defining an information database and an incorporative protocol between source and destination. Because the X server is shared by all applications displayed at the workstation, they can use the server as a conduit to pass information among one another. Actually, X provides a general mechanism, *properties*, for sharing of information among different applications. A property is a collection of named data items. Applications in X can communicate by hanging properties on windows, as if the properties were tags. Any application can come along and write information on any property, or can read any property. The X server does not show the properties to the user, but merely stores them so applications can share them. Every property has a name associated with it. A property name is an arbitrary character string. The X server keeps each window's properties separate from those of all other windows. Each property also has a data type. Before communicating via properties, applications must agree about the name and type of the property they will use to pass information. Applications must also agree about the window on which the property they will use to pass information.

We decided to use the property with the root window as a shared database. The data consists of a collection of widget identifiers of all instances of potential destination objects that are created by applications. This database is built through the procedure of "destination registration" that is a part of the methods in the Simple widget class. Since this property resides in the root window associated with the X server, it is visible to every application.

Whenever an instance of a destination widget class (with its *droppable* property as *True*) is created, it registers to the database with its widget identifier, which is unique for every widget. This is done by having the object call a method which sets a property associated with the root window. Similarly, whenever a destination object is destroyed it deletes its identifier from the database. In this way the database is built dynamically and always reflects the identifiers of the most current dropsites. When a source is dragged, it calls a method in the widget class that extracts all current widget identifiers from the database. This is called binding. Once the client has all the widget identifiers it can easily retrieve the geometry and other information from the X server and know which dropsite the pointer is located in. Since the database is located in the root window, this scheme can support drag-and-drop across two different applications. Once the source object has the identifier of the dropsite, a *ClientMessage* event can be sent to the destination with source's own identifier. In this way, a two-way communication between the source and destination can be created.

The following pseudocode for the draggable and droppable feature added in Simple

widget class:

```
/* The droppable part of the implementation: */

    IF a widget is droppable

        - At the creation: pre-register its widget ID
        - At the deletion: withdraw its registration
        - Solicit ClientMessage event for messages from source
        - After receiving message from a source:
            - Identify the source, and receive the data
            - Invoke the dropCallback installed by user
            - Acknowledge the source

/* The draggable part of the implementation: */

    IF a widget is draggable

        - When a pre-assigned mouse button is pressed and dragging:
            - Get widget IDs of potential dropsite
            - Check the current mouse position
        - When the mouse button is released for dropping:
            - Identify the destination, if it is droppable:
                - invoking the dragCallback for source data
                - Send a message to the destination with the data
            - Receive the acknowledgment
```

Figure 3: Pseudocode for Drag-and-Drop Operations in Simple

5 Testing

For testing purpose, we implemented the drag-and-drop feature in the List widget, which is a sub-class of the Simple class. Also, we implemented an icon widget class (called Ficon for Function Icon). The icon widget inherits the drag-and-drop feature from its superclass Simple widget, and also provides resources for customizing its appearance, such as its pixmap and label. With our approach, implementing the drag-and-drop in rest of the subclasses of the Simple widget becomes trivial. All the implementation details are transparent to users. All that a user needs to do is to set the corresponding draggable and droppable resources, and to install the drag and drop-callbacks. Since resources are used in setting the drag-and-drop operation, a resource file can be used to change these settings without re-compile the program.

To test our implementation, we made a simple file manager. As shown in Figure 4, two list widgets (in the left) and five icon widgets (in the middle and right) are created in our testing. The list widget in the upper left (List1) contains a list of file names and is only set *True* to *draggable*, whereas the list in the down left (List2) contains a list of directories and is set *True* to both *draggable* and *droppable*. The drag-callbacks in List1 and List2 set the

source data to be the list that is currently selected in the widget (the selection operation is one of the features provided by the List widget class; in our testing, we only use the single selection). The drop-callback in List2 set the destination also to be the currently selected list.

The five icon widgets are customized with their own labels and pixmaps. Each of them is set *True* to *draggable* and the drag-callback sets the source data to be the widget ID of the icon. Only three icons: "XEDIT", "FILELIST", and "QUIT", are set *True* to *droppable* and their drop-callbacks set the destination functions as invoking a simple X editor for a named file of the source data, displaying a window with the directory contents of the source data, and destroying a widget with the ID in the source data, respectively.

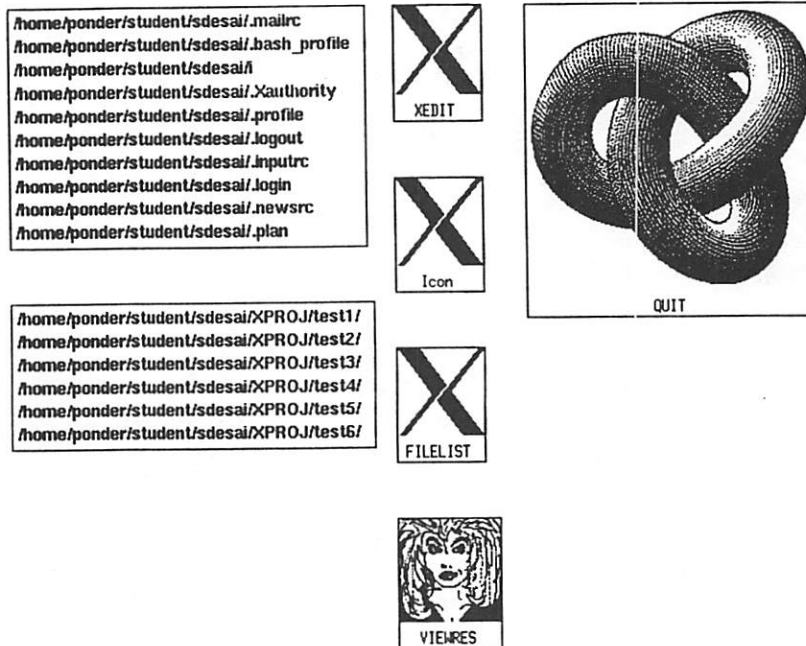


Figure 4: Screen Dump of the Testing Drag-and-Drop Application

In our testing, files (in List1) can be dragged and dropped to directories (in List2) as a copy operation, files can be dropped on an icon drop-site (in XEDIT) invoking edit operation on the file. Directories (in List2) can be dragged and dropped to itself (List2) as a copy operation to other directories. Directories can also be dragged and dropped on an icon (FILELIST) to show the files contained in the directories. An icon widget can be destroyed by dragging and dropping into the QUIT icon. In the same way, a file or a directory can be dragged from their list widget and be destroyed by dropping it into the QUIT icon (some sort of polymorphism is supported there).

The drag-and-drop feature worked without any problem in both List and Icon widgets. Overall, the whole system behaved well with almost instantaneous response time. Even with as many as thirty source and destination objects the time taken was much less than the tolerable limit. The drag-and-drop operations also worked across different applications. The most interesting point is that all these various forms of drag-and-drop can be achieved with the single framework of generalized drag-and-drop.

Figure 5 shows the sample code of our testing program. This demonstrates that with the support of generalized drag-and-drop, implementation of drag-and-drop operations in

a user application becomes simply a matter of setting some resources and callbacks in GUI objects.

```
/* Declare all necessary widgets */

Widget toplevel,form_widget,list1,list2,ficon1,ficon2,ficon3;

/* write necessary callback routines */

void l1_drop_callback(w,client_data,call_data)
Widget w;
caddr_t client_data;
XawDropReturnStruct *call_data;
{
/* Code for drop operation of List1 */
}

void l2_drag_callback(w,client_data,call_data)
Widget w;
caddr_t client_data;
XawDragReturnStruct *call_data;
{
/* Code for drag operation of List1 */
}

main(argc, argv)
int argc;
char *argv[];
{
/* Initialize the shell widget here */
...

/* Create new widget instances */

n = 0;
XtSetArg(args[n],XtNdragable,TRUE);n++;
...
list1 = XtCreateManagedWidget("List1",
listWidgetClass,form_widget,args,n);

n = 0;
XtSetArg(args[n],XtNdropable,TRUE);n++;
XtSetArg(args[n],XtNdragable,TRUE);n++;
...
list2 = XtCreateManagedWidget("List2",
listWidgetClass,form_widget,args,n);

n = 0;
...
ficon1=XtCreateManagedWidget("Ficon2",
FiconWidgetClass,form_widget,args,n);

n = 0;
...
}
```

```

ficon2=XtCreateManagedWidget("Ficon2",
                             FiconWidgetClass,form_widget,args,n);

n = 0;
...
ficon3=XtCreateManagedWidget("Ficon3",
                             FiconWidgetClass,form_widget,args,n);
...

XtAddCallback(list1,XtNdropcallback,l1_drop_callback,NULL);
XtAddCallback(list2,XtNdropcallback,l2_drag_callback,NULL);
...

XtRealizeWidget(toplevel);
XtMainLoop();
}

```

Figure 5: Sample Code of a Drag-and-Drop Application

6 Conclusions

Due to its interactive and visual nature, we argue that the operation of drag-and-drop should be an intrinsic part of any GUI systems. Unfortunately, most of current GUI systems treat drag-and-drop as an exception rather than a norm. This leads to difficulty and complexity in implementing operations of drag-and-drop in user applications. This paper presents a framework of generalized drag-and-drop that integrates the drag-and-drop operation as part of a GUI system and supports various forms of drag-and-drop in a broad range. A prototype implementation in X window system shows that the framework is both feasible and beneficial. Certainly, more issues need to be addressed before this scheme can be applied in real production systems, including the definition of drag-and-drop semantics in various GUI objects, the support of polymorphism for different data types in the operation, and the design of a complete protocol.

References

- [1] M.S.Ackerman, D.Converse, and R.R.Swick, "Widget Internals," *Tutorial Notes of 6th Annual X Technical Conference*, January 1992.
- [2] Alan Southerton, and Steve Mikes, "Friendly Desktops", *UnixWorld*,8(9), Nov. 1991.
- [3] Apple Computer, Inc., "Inside Macintosh," Vol. III,1988.
- [4] Apple Computer, Inc., "Macintosh System Software User's Guide," Version 6.0, 1991.
- [5] P.A.Asente and R.R.Swick, "X Window System Toolkit," Digital Press, 1991.
- [6] G. Begeg-Dov and E. S. Cohen, "Implementing Drag & Drop for X11," *The X Resource*, Issue 1, Winter 1992, pp. 169-190.

- [7] J.D.Foley, A.van Dam, S.K.Feiner, and J.F.Hughes, "Computer Graphics Principles and Practice," 2nd Ed., Addison-Wesley, 1991.
- [8] J. McCormack and P. Asente, "Using the X Toolkit or How to Write a Widget," *Proc. of the Summer 1988 USENIX Conference*, 1988, pp. 1-13.
- [9] S. Mike, "New Ways to Program in Object-Oriented X," *UNIX World*, Vol. 8, No. 8, August 1991, pp. 103-108.
- [10] Sun Microsystems, Inc., "OPEN LOOK Graphical User Interface Functional Specifications," Addison-Wesley, 1989.
- [11] Neuron Data, "Open Interface Reference Manual, Version 1.03," Neuron Data, Inc., 1991.
- [12] Open Software Foundation, "OSF/Motif Programmer's Guide," Release 1.1, Prentice Hall, 1991.
- [13] D. Smith, C. Irby, R. Kimball, and B. Verplank, "Designing the Star User Interface," *Byte*, Vol. 7, No. 4, pp. 242-280, April 1982.
- [14] J. D. Smith, "Object-Oriented Programming with the X Window System Toolkits," John Wiley & Sons, 1991.
- [15] R. Swick, and M. Ackerman, "The X Toolkit: More Bricks for Building User Interfaces," *Proc. of the Winter, 1988 USENIX Conference*, pp. 221-233, 1988.
- [16] D. A. Young, and J. A. Pew, "The X Window System Programming & Applications with Xt, OPEN LOOK Edition," Prentice Hall, 1992.

The Xt Intrinsic as a General Purpose Application Development Platform or A User Interface Toolkit with Optional Users

Jordan M. Hayes
Heuristicrats Research, Inc.
jordan@Heuristicrat.COM

Charles A. Ocheret
Investment Management Services, Inc.
chuck@IMSI.COM

Abstract

Many graphical user interface (GUI) toolkits provide an event-driven framework which dispatches control to specific pieces of code based on an event stream, such as mouse clicks or keystrokes. These toolkits have incorporated utilities into their main loops for dealing with synthetic timers, I/O multiplexing, and idle procedures. Other real world problems fit this event-driven model quite well; database servers, communications line handlers, and interpreter front ends, to name a few. With only a few exceptions, we've found the Xt Intrinsic to be an ideal platform for building a distributed real-time¹ market data system, especially the non-GUI portions², despite the fact that this toolkit was principally designed to be used as a GUI toolkit. This paper describes our success with this approach and the limitations we encountered.

Introduction

Without first-class thread support from the operating system, event-driven programming must be implemented using non-blocking multiplexed I/O, synthetic timers, and cooperative long-running processing blocks, often referred to as coroutines. Several GUI toolkits have incorporated event handling and dispatching into their processing model to deal with mouse clicks, keystrokes, and other asynchronous events. Sun's XView has the notifier, Tk has `Tk_MainLoop()`, OI has its own main loop as well. There is a tendency for programmers to reinvent large amounts of this GUI toolkit infrastructure when implementing non-interactive event-driven applications. Any application that must deal with more than one source of input must respond with minimum latency in an asynchronous fashion, much like their graphical counterparts. This infrastructure is often developed by a different group of programmers than the GUI folks in a large project, and can lead to conflicts that must be resolved through no small amount of hackery. Additional programmer cycles are spent on such mundane things as command line argument processing, user and system-wide customization of runtime attributes, the build environment, operating system portability, and even language issues (K&R C vs. ANSI C vs. C++).

An alternative is to identify an existing platform which provides basic support for event-driven programming, and select this as the substrate upon which to build the modules necessary for the system. We have found that such an environment is provided by the sample implementation of the X Window System. Despite ostensibly being a set of cooperating GUI toolkits (for a particular window system even), much of the structure and mechanism found in Xlib and Xt do not require the opening of an X Window display connection. A few do require a display connection, but for no good reason.

¹ Wall Street's version of "real-time," not the "real-time" of Process Control.

² Certainly Xt helped quite a bit in the implementation of a widget set, but ...

We realized early on that while we certainly could provide a *NULL server*³ for these applications to attach to, or even designate dedicated hardware that would always be available (after all, we were working on Wall Street), those solutions were clearly unsatisfactory. In a sense, there is a hidden toolkit-within, waiting to be discovered, and we decided to search for it. If the non-GUI components of our platform were built with Xt, combining them with user interface objects (Widgets, etc.) later would be much easier; the integration job was nearly non-existent. In a sense, this is the overriding goal of any toolkit: to write code once, debug it well, and use it everywhere that is appropriate. Having access to the X source code was also a major benefit.

Abstraction Layer / Portability Issues

Like the poor hacker in a new job, the designers and implementors of the X and Xt Intrinsics libraries were faced with not being able to count on any of the support they required for event processing; indeed, they were unable to even determine whether to use `select()` or `poll()` for I/O multiplexing. Similarly, synthetic timers and coroutines would be required, but no standard implementation existed. Because of this the libraries provide such an implementation and an extra level of portability was made available, with general purpose interfaces defined.

Xlib provides quite a few useful constructs to both increase productivity and portability. For instance, it provides an excellent open-ended string hashing mechanism which it uses internally to reduce the number of string comparisons. Each unique string is converted to an integer, called a *Quark*. We used this to build a simple but powerful bucket-based hash table. Time/space tradeoffs can be made in a straightforward way, since the *Quark* mechanism assigns integers in a serial fashion, filling the buckets evenly. As a special case in one application that requires hashed access to pointers keyed by 300,000 strings, we simply use an array of pointers indexed by the *Quark* to provide a single indirection from *Quark* to pointer, saving both space in the table and overhead in search time; each "bucket" is one-deep, and (almost) all buckets are filled.

Xt builds on what Xlib provides and adds a number of useful features of it's own. The "arbitrary pointer" type is determined through the C preprocessor (a fierce-AI) in header files with the macro `XtPointer` defined as `"void *"` or `"char *"` as appropriate. Similarly determined macros are defined for portably allocating a structure (`XtNew(type)`), copying strings (`XtNewString(str)`), counting the number of elements in a static array (`XtNumber(array)`), and computing the offset of a structure element (`XtOffsetOf(type, element)`). Cover functions for memory allocation (`XtMalloc()`, `XtCalloc()`, `XtRealloc()`, as well as `XtFree()`) provide common answers to those annoying questions like:

"What does `malloc(0)` return?"

"What happens if I pass a *NULL* pointer to `free()` or `realloc()`?"

"What should I do if `malloc()` fails?"

Commonly used typedefs are provided for *String*, *Boolean*, and the macros for the constants *TRUE* and *FALSE* are defined.

The use of Imake for generating portable makefiles and careful adherence to conventions established by the X Consortium vastly simplified the process of porting our applications to numerous platforms. Since X is available on a wide variety of operating systems and hardware, from Macs to Crays, we could be sure that the difficulty of porting to a new environment would be minimized, and that there would be clear guidelines to help us. We were not headed off *where no man had been before*. More than 250,000 lines of library and application code were ported from Solaris 1.x to IRIX (SysVR3 at the time) and Solaris 2.3 (SysVR4) in about a programmer-day each. Switching from the bundled Sun K&R C compiler to ANSI C and even C++ required only tidying up a few warnings.

³ Such a server would respond to the X protocol but not be physically attached to an actual frame buffer.

Imake in particular was an enormous help, since decisions about if and how shared library support is provided, whether or not ndbm is provided, whether or not to build CodeCenter rules into the makefiles, as well as compiler flags for all supported systems (how about that “-Wf,-XNp9000,-XNd8000,-XNh2000” flag for IRIX!) are transparent to the cross-platform developer. Gnu Configure handles some of these problems in quite a different manner, but is quite incomplete compared to Imake’s coverage.

We believe that this extra attention to portability has also resulted in tighter code with fewer problem areas; portable code is often better code. Where portability questions arose, there was a large solid body of examples to examine for inspiration. Finally, as much as we dislike the use of signals as a means of IPC in our applications, they are in some instances a fact of life. X11R6 adds POSIX signals to the set of event sources that can be integrated into the event stream safely. For simple usage (catch SIGINT, clean up, and die), this addition should prove to be quite useful. It remains to be seen what effect it will have on Joe Coder who may choose to use it to (poorly) reinvent multiplexed I/O through catching SIGIO⁴.

The Main Loop

Event driven applications all begin with the same steps: establish any initial event sources and fall into a “main loop” that arranges for events to trigger callback procedures. Each callback is expected to carry along whatever state is needed to avoid the use of global variables. This state is often represented as a pointer to arbitrary allocated memory; Xt often calls this “closure” and uses XtPointer as it’s type. Since callbacks are non-preemptive, the callback writer should make a decision about how much processing time is required by the callback and perhaps perform only part of the task, rescheduling itself for another timeslice later. The main loop is responsible for coordinating these “internally generated events” and dispatching them. In addition, the main loop is responsible for dispatching externally generated events such as file descriptor readiness. In Xt, this function is called XtAppMainLoop()⁵, and the simplest of Xt applications looks like this:

```
#include <X11/Intrinsic.h>

main()
{
    XtAppContext  app;

    XtToolkitInitialize();
    app = XtCreateApplicationContext();
    XtAppMainLoop(app);
}
```

This program initializes the Xt Toolkit, creates an application context, and drops into the main loop. There are no event sources defined, so this program will never do anything, but you have to crawl before you can walk.

Synthetic Timers

One important event source is synthetic timers⁶, which can be used to schedule processing to take place at some point in the future. XtAppAddTimeout() is the function used for this purpose in Xt. It takes as arguments the application context, a number of milliseconds to wait, a callback function, and closure, and returns an XtIntervalId that can be used to cancel the timer event before it goes off.

⁴ Cynical, aren’t we? Perhaps just battle-scarred.

⁵ The “App” in XtAppMainLoop() stands for “Application Context,” a token that is used to coordinate all the event sources, a sort of meta-context for the entire application.

⁶ Most UNIX implementations provide access to only one high-resolution timer, necessitating building your own timer mechanism, typically by keeping a sorted list of pending timers and using the closest to expiration to determine the final argument to select().

The following program fragment sets up a timer that decrements a count, cleaning up when the count reaches zero. Each decrement reschedules itself with a new timer in the callback routine `_DoCounter`. The callback function looks like this:

```
typedef struct Counter {
    XtAppContext app;
    XtIntervalId id;
    int count;
} Counter;

/*ARGSUSED*/
static void
_DoCounter(closure, id)
    XtPointer closure;
    XtIntervalId *id;
{
    Counter *c;

    c = (Counter *)closure;
    if (--c->count == 0)
        /* no more counting; clean-up */
        XtFree((char *)c);
    else
        /* reschedule */
        c->id = XtAppAddTimeOut(c->app, 1000L, _DoCounter, (XtPointer)c);
}
```

Somewhere in the application, this task would get initiated with code similar to this:

```
Counter *c;

/* ... */
c = XtNew(Counter);
c->app = app;
c->count = 10;
c->id = XtAppAddTimeOut(c->app, 1000L, _DoCounter, (XtPointer)c);
/* ... */
```

Timer events can be canceled via `XtRemoveTimeOut()`, passing the `XtIntervalId` returned from `XtAppAddTimeOut()`. One extension to this mechanism we built is an object that implements exponential backoff using these timers called, oddly enough, a `Retry`. You create a `Retry` object specifying an initial time interval to retry, a terminal time interval, a callback function to invoke each time the timer fires, and a closure. Both time intervals are expressed in milliseconds, to match Xt's notion of time. A sample usage looks like this:

```
static Boolean
_AttemptOpen(closure)
    XtPointer closure;
{
    _OpenState *os;

    os = (_OpenState *)closure;
    if ((os->fp = fopen(os->filename, "r")) != (FILE *)NULL) {
        os->retry = (Retry)NULL; /* retry is destroyed upon return */
        return(TRUE);
    }
    return(FALSE); /* backoff, then try again */
}

/* ... */
os->retry = RetryCreate(os->app, 0, 30000, _AttemptOpen, (XtPointer)os);
/* ... */
```

This code fragment would try to open a file immediately, but if it failed, it would try again after an amount of time that starts off at zero and makes it's way to 30 seconds. When the interval reaches 30 seconds, it will stay there. If the open succeeds, the function returns TRUE and the `Retry` object is cleaned up. To cancel this activity, call `RetryDestroy(os->retry)`. Another application of this technique can be found in code that tries to establish a socket connection to a network service. If the server is down, you don't want an application to spin its wheels trying to connect, especially if there are two hundred similar clients on your network trying to do the same thing. A random skew is added to the timer interval to avoid sympathetic behavior in the case of a server restart. If the server takes long enough to restart that all clients have backed off to the maximum interval, you want to make sure that as soon as the server is available to accept connections that a high proportion of clients do indeed get connected. Due to the way `listen()` is implemented in many UNIX systems, often only 8 connection requests can get queued on the server. Staggering the timers even small numbers of milliseconds can greatly increase the likelihood that clients can connect successfully in the first timer cycle after the server is ready. The risk you run of having a subset of users connect while the rest wait for a (maximal) timer cycle before connecting can be detrimental to your health (not to mention year-end compensation) on a crowded trading floor full of unsympathetic users. At the rate of 8 per cycle, you could have an angry mob on your hands after a few minutes. Where time is money, this is quite important to implement correctly.

I/O Selection and Multiplexing

File descriptors can be added to the set of event sources with `XtAppAddInput()`. Callbacks can be set to fire when input is available to be read, buffer space is available to write, or exceptional conditions occur. Since data arrives asynchronously and from multiple sources, it is possible that message boundaries are not preserved. Input handling routines must be able to accept whatever amount of data is available and return to the main loop even if a full message has not been received⁷. Blocking in any system call is not permitted, as a failure in any source could deny service to the rest of the application. Similarly for writing, blocking cannot occur, and the source must provide buffering until all bytes are written. Building on this, we implemented a protocol engine for TCP sockets. We used ONC/XDR as a marshaling mechanism for the messages. Non-blocking reads and writes are performed to avoid any particular source or sink from monopolizing a single-threaded application. On the server side, the listen socket is added to the event stream, waiting for a read event. When a connection is attempted, On the client side, a non-blocking `connect()` can be made with a callback firing when the connection is actually established or refused.

It is interesting to note that tools like yacc and lex have some problems integrating nicely with an environment that provides asynchronous I/O. In particular, a typical program that uses a yacc parser calls `yyparse()` which synchronously calls the lexical analyzer to get a token. It continues to call the lexical analyzer until a complete grammar is recognized. The lexical analyzer in turn synchronously calls the I/O routines repeatedly until it has enough input to return a token. This leads to difficulty because a call to `yyparse()` can lock out the rest of the application until enough input is found.

We found it necessary to invert the lex/yacc model. `XtAppAddInput()` is used to register a routine to be called when input is available. That routine reads what data is available and then calls the lexical analyzer with the new data. The lexical analyzer examines the data for tokens and invokes the parser which each token found. If the lexical analyzer can't find an entire token at the end of the buffer, it returns, leaving the unconsumed bytes in the buffer to be appended to when the input handler fires again. Since the parser is invoked one token at a time it must remember its state from token to token. Similar but incompatible substitutes for lex and yacc were written to support this processing model.

Cooperative Long-running Tasks

⁷ Incidentally, Xlib does this incorrectly when processing the X protocol stream. Perhaps Xlib should be reimplemented on top of Xt. :-)

One of the most often asked questions from novice GUI programmers is how to continue handling events while performing some processing that can take a long time. A reasonable approach involves splitting up the task into small bite-sized pieces and executing a portion of it each time there is idle time available. In theory, there is no such thing as idle time because as soon as you have determined that there are no events pending, and begin your idle time processing, events can arrive. About the best you can expect to do is take your timeslice, do some of your processing, and return to the main loop. This adds some overhead to the total time to execute the task, but the response curve is smoother and allows an easy way to interrupt the task. Xt provides a way to get some idle time through `XtAppAddWorkProc()`, which takes a callback and some closure, and returns an `XtWorkProcId` which can be used to cancel the work proc with `XtRemoveWorkProc()`. The callback is fired when the main loop determines that there are no other events pending.

An example of where we made good use of this facility is in a database server that delivers stock quotes to applications. Sometimes the queries can generate megabytes of data, which can't all be delivered at once due to network buffering. Many seconds could go by before another query could be handled if the request was handled synchronously. Our protocol engine has hooks to determine how many messages are presently queued waiting for buffer space in the socket to become available, so we could use this to throttle the producer of, say, 100,000 messages. We might make a determination that 100 messages was a maximum that we want to have outstanding at any given time; since they get buffered by the system, you could easily run out of memory with just a few customers, especially if their network is congested or slow - like if they are in Paris or London and the database is in New York.

The work procedure would look something like this:

```
typedef struct Delivery
{
    Client      c;
    Database    d;
    int         todo;
    int         current;
    XtWorkProcId id;
} Delivery;

static Boolean
_DeliverSomeMore(closure)
    XtPointer      closure;
{
    Delivery      *del;

    del = (Delivery *)closure;
    while (ClientMsgsQueued(del->c) < 100) {
        for (i = 0; i < 100 && i + del->current < del->todo; i++)
            ClientDeliverMsg(del->c,
                             DatabaseMsg(del->d, del->current + i));
        del->current += 100;
        if (del->current > del->todo) {
            /* cleanup del */
            return(TRUE);          /* all done, proc unregistered */
        }
    }
    /* more to do next time we can */
    return(FALSE);
}
```

The outer while loop makes sure that if the network can actually handle this kind of throughput that we take advantage of it - 100 might not be enough to satisfy an FDDI-equipped SGI; you might actually want to instrument this to report how useful 100 is as an estimate of a correct amount of work to do. In the query processing section, we might see some code that looks like:

```
Delivery      *del;
```

```

del = XtNew(Delivery);
del->c = c;
del->d = d;
del->todo = DatabaseNumItems(d, query);
del->current = 0;
del->id = XtAppAddWorkProc(app, _DeliverSomeMore, (XtPointer)del);

```

Customization and Resources

Xt provides utilities for handling command line arguments with system, user, class, and instance defaults. Customization is handled with resources, which are arbitrary key-value pairs. The value can be of arbitrary type with a number of converters supplied for the common cases, such as `String` to `Boolean` and `String` to `Pixel`. Because the converters sometimes require access to an X display (for instance, in generating a pixel value), we had to have an alternate interface that didn't require the display.

First we define a data structure that contains a place for the resources to get stored:

```

typedef struct Barney {
    XtAppContext  app;

    /* resources */
    String        song;
    int           hugs;
    Boolean       purple;
} Barney;

```

Then we define which resources we will allow the user to set. Resources are represented with an `XtResource` data structure that looks like this:

```

typedef struct _XtResource {
    String        resource_name;      /* Resource name */
    String        resource_class;     /* Resource class */
    String        resource_type;      /* Representation type desired */
    Cardinal      resource_size;      /* Size of representation */
    Cardinal      resource_offset;    /* Offset from base for value */
    String        default_type;       /* representation type of default */
    XtPointer     default_addr;       /* Address of default resource */
} XtResource, *XtResourceList;

```

For example, we might have three resources:

```

#define offset(field)      XtOffsetOf(Barney, field)
static XtResource  resources[] = {
    { "song", "Song", XtrString, sizeof(String),
      offset(song), XtrString, "I Love You" },
    { "hugs", "Hugs", XtrInt, sizeof(int),
      offset(hugs), XtrString, "17" },
    { "purple", "Purple", XtrBoolean, sizeof(Boolean),
      offset(purple), XtrString, "TRUE" },
};
#undef offset

```

Then we provide a description of what the command line arguments are and what additional arguments might be required. This is sort of a `getopt()` on steroids.

```

static XrmOptionDescRec options[] = {
    { "-song",      ".song",      XrmoptionSepArg,  NULL },
    { "-hugs",      ".hugs",      XrmoptionSepArg,  NULL },
    { "-purple",    ".purple",    XrmoptionNoArg,   "FALSE" },
};

```

In our example, the resource "hugs" is an integer which defaults to 17, and the "-hugs" command-line argument takes a separate argument which is the value. The following program would parse command line arguments and print the value of hugs.

```
main(argc, argv)
{
    int     argc;
    char    **argv;

    static Barney b;

    XtToolkitInitialize();
    b.app = XtCreateApplicationContext();
    ResourceParse(argv[0], resources, XtNumber(resources),
                  options, XtNumber(options), (XtPointer)&b, &argc, argv);
    (void)printf("hugs = %d\n", b.hugs);
}
```

Output from this program might look like this:

```
% a.out
hugs = 17
% a.out -hugs 27
hugs = 27
```

Files can be provided that also assign values to resources. Such a file could look like:

```
*hugs:      14
*purple:    FALSE
```

Most often these files serve as application defaults files which can be overridden based on user, host, X display, and application invocation name.

Object Oriented Programming

The Xt Intrinsics provides a widely used framework for doing object oriented programming (OOP) in C, allowing for single inheritance⁸, encapsulation, messaging, and operator overloading. Xt provides an abstract base class called `Core` from which all other widget classes are derived. `Core` provides the base functionality used by all user interface entities that require a window. For example in the Athena widget set, `Command` is a subclass of `Label` which in turn is a subclass of `Simple` which finally is a subclass of `Core`.

Each class implementation defines two structures, a class record and an instance record. There is one class record⁹ per class which describes the behavior of the class. Each instance of the class gets its own instance record which records the state of the instance. Inheritance is accomplished in a manner familiar to many C programmers trying to do OOP. The class and instance records consist of an ordered group of subparts corresponding to the classes in the hierarchy. For example, the class and instance records for the Athena `Command` widget look like this:

```
typedef struct _CommandClassRec {
    CoreClassPart      core_class;
    SimpleClassPart    simple_class;
    LabelClassPart     label_class;
    CommandClassPart   command_class;
} CommandClassRec;

typedef struct _CommandWidgetRec {
```

⁸ There are roundabout ways to accomplish multiple inheritance within the Xt context but this is neither the time nor the place; we have a paper in progress to explain one such technique.

⁹ This is sometimes referred to as the "factory object" in other OOP systems.

```

        CorePart          core;
        SimplePart        simple;
        LabelPart          label;
        CommandPart        command;
    } CommandWidgetRec;

```

Using this scheme, a function that expects a pointer to a record for class A can be passed a pointer to a record for any subclass of A and still work. This is because the component parts for class A and all of its superclasses are at the same offsets within the record. In fact, nearly all application code refers to objects by the type `Widget` which is a pointer to a structure containing only a `CorePart`. For example, creating an Athena Label widget might look like this:

```

Widget w;
...
w = XtVaCreateWidget("myLabel", labelWidgetClass, parent,
                    XtNLabel,      "Hello World",
                    XtNjustify,    XtJustifyCenter,
                    NULL);

```

The Xt Intrinsics also introduce a convention for encapsulating private information. The source code for a class implementation is divided into three files: a public header file (the `.h` file), a private header file (the `P.h` file), and an implementation file (the `.c` file). For example, the `Command` widget class can be found in `Command.h`, `CommandP.h`, and `Command.c`. The public header file reveals nothing about the internals of the class implementation including the class hierarchy. It only declares constants and external interfaces to the class. The private header file contains details of the new parts of the class and instance records. This file is used by the implementation and by subclasses. This represents a significant advantage over C++ which requires you to place the private, protected, and public information in the same header file. When some detail of the private members changes, all consumers of a C++ class need to be recompiled even if they don't have access to the private members. A Widget implementor can change the details of the private header files and only worry about subclasses recompiling. For large projects like ours the productivity hit of massive C++ recompiles would have been significant.

Class methods can be provided via function pointers placed inside of a class record. These are roughly equivalent to the virtual functions of C++. These are initialized to default values in the implementation file and can be overridden in a subclass' implementation file. A special initialization value `XtInherit` is provided to give subclasses the opportunity to copy function pointers from their superclasses. Public interfaces to these methods take a pointer to an object, indirect to find the class record and invoke the appropriate function contained within. For example, a simple drawing widget (let's call it a `Drawer`) might want to publicize a method to draw a line segment. The private header file (`DrawerP.h`) would declare a pointer to the method as follows:

```

typedef void (*DrawerLineProc)(
    #if NeedFunctionPrototypes10
        Widget w,      /* should be a Drawer widget or a subclass */
        double x1,     /* first point */
        double y1,
        double x2,     /* second point */
        double y2
    #endif
);

typedef struct _DrawerClassPart {
    ...
    DrawerLineProc    line;
}

```

¹⁰ This macro is one of the conventions to follow strictly so that Imake can automatically make K&R C, ANSI C, and C++ be happy.

```
...
} DrawerClassPart;
```

The public header file (`Drawer.h`) would declare a public interface to the method as:

```
extern void DrawerLine(
#ifdef NeedFunctionPrototypes
    Widget w,      /* should be a Drawer widget or a subclass */
    double x1,     /* first point */
    double y1,
    double x2,     /* second point */
    double y2
#endif
);
```

The implementation of `DrawerLine` in the implementation file (`Drawer.c`) could be:

```
void
DrawerLine(w, x1, y1, x2, y2)
    Widget w;
    double x1, y1, x2, y2;
{
    DrawerWidgetClass wc;
    DrawerLineProc proc;

    if (!XtIsSubclass(drawerWidgetClass)) /* not any old widget */
        return;
    wc = (DrawerWidgetClass)XtClass(w); /* find widget's class */
    proc = wc->drawer_class.line; /* find method for class */
    if (proc != (DrawerLineProc)NULL)
        (*proc)(w, x1, y1, x2, y2);
}
```

After making sure that the widget passed in is a `Drawer` or a subclass of `Drawer`, we invoke the method found in the class record of the widget passed in. The way the actual method gets into the class record is through static initialization in `Drawer.c`. Now suppose a subclass of `Drawer`, let's call it `TopDrawer`, is implemented which needs to override the behavior of `Drawer's` line method. All it needs to do is to statically initialize the `TopDrawer` class record with a different line method in the `DrawerClassPart`. This explicit control makes it convenient to model all sorts of method behavior. Methods that chain superclass-to-subclass, subclass-to-superclass, and inheritance can be easily accomplished.

Non-Windowed Objects

In later releases the X_t architects realized that their OOP system could be useful for creating objects that did not correspond to windows on the screen. They introduced new abstract base classes called `Object` and `RectObj` as ancestors of `Core`. However, this leads to a problem in preserving binary compatibility since `Core` class expects to find the `Core` part at the start of the nested structure. They played a trick by making `Object` and `RectObj` occupy the same position as `Core`, but stubbed out fields which would not be used. This allows for creation of classes which have nothing to do with the user interface.

The fatal flaw of this scheme is that X_t Objects still require that a connection to an X Windows display exists in order to be created. This is a problem since we wish to create some stand-alone daemon processes with as few dependencies on the external environment as possible. All we need is to have to tell our boss that the database is down because someone logged out of the master X display; that would just make our day. X_t Objects need the X server because of the way the resource database is built from a property on the root window.

Using the OO paradigm presented by Widgets, we have implemented non-windowed objects that are not related to any of the Xt supplied base classes, called `DoDads`. `DoDads` subclass from an abstract base class called `Eye`. Like Widgets, `DoDads` support single-inheritance, encapsulation, messaging, operator overloading and the resource model. As much as possible, the `DoDad` infrastructure shares source code with that of Widgets. A number of enhancements were made where we felt we could be more clever than the Intrinsics, and it became necessary to provide alternative interfaces for common tasks (i.e. `DoDads` and Widgets are not interchangeable in their APIs). For example, you must call `DoDadCreate()` instead of `XtCreateWidget()`. Programmers who are familiar with Widgets can be fluent with `DoDads` in a short period of time.

As an example of where this has proven productive we have used `DoDads` to create an abstract base class called `LineReader`. This object knows how to accept input from a socket, serial port, HDLC port, or a regular file. It supports logging all information passing through it. This class was easily subclassed to provide readers for specific datafeeds. The availability of the resourcing mechanism to `DoDads` proved invaluable in customizing `linereader` configurations. For example, we developed a reader which in production would read data from a terminal server via a socket. However, to test the reader against new prototype versions of the datafeed we had to configure the reader to dial out through a local modem and then speak a bogus protocol through the serial line. The configuration file has sections for each instance of the application, with one defining resources to access a socket while another has resources for the modem.

Conclusion

The Xt Intrinsics provide a rich set of utilities which we have used to build large, complex, distributed systems that do not contain user interfaces. The designers of Xt were faced with a requirement for mechanisms across a large set of disparate operating systems that simply weren't there. Other GUI toolkits have attempted to provide the same abstractions and were similarly disappointed with the lack of provided mechanisms for timers, I/O multiplexing, main loops, user and system level customization of runtime attributes, and the like. TCL/Tk, OI, InterViews and XView all have similarities and could have been used in our platform. We chose Xt as a base because we knew that we would be building our GUI components with Xt-style Widgets, but similar results could have been accomplished with these other toolkits. Perhaps this points to a failing of these toolkits, that they did not completely separate out the non-GUI components from the GUI components (each of these have their quirks that need to be dealt with in the non-GUI case); we could take the time to do it right, but hey, we have work to do.

Acknowledgments

Much of the work represented in this paper is the result of several years work at various jobs by the authors, most recently honed and implemented at Investment Management Services, Inc. While actually building production systems with these platforms provided the majority of breakthroughs and insights, the authors nevertheless gratefully acknowledge the impact of intense conversations, long-distance rambling e-mail, and all-night design and coding sessions with Dan Fisher and Doug Kingston of Morgan Stanley & Co., Inc., Ron Turner and Oleg Schattoff of Market Vision, Inc., Mike Curry of Teknekron Communications Systems, Inc., Donna Converse of the X Consortium, Inc., and Othar Hansson from Heuristic Research, Inc. The concepts presented here appear in several toolkits implemented over the past few years, but no single package is available due to proprietary information guidelines.

Bridging the Technology Generation Gap: Upgrading a Network Management Application to a New Technology Base

Jay S. Lark

Teknekron Communication Systems, Inc.
jlark@tcs.com

Abstract

NMCS is a network management system which has been developed to manage dedicated fractional-T1 networks, allowing end-users such as banks and government agencies the ability to dynamically manage the allocation of their dedicated communication facilities. The first version of NMCS was delivered on a PC platform; it was a successful product but could not adapt to meet new customer demands.

NMCS II is a significant upgrade to the first NMCS product. In addition to extending its functional capabilities, NMCS II is built on a different technology base from its predecessor: the hardware and operating system, application architecture, user interface, and database have all been upgraded to a new technology generation. It is the most sophisticated application of its kind in commercial deployment.

NMCS II is now installed and operational at several customer sites throughout Canada. While the product has passed its formal acceptance, a number of acceptability issues were present in its initial release. Many of these issues derive from the changes to the technology base, which we call the "technology generation gap."

This paper describes the NMCS II system at a high-level, and lists the technology generations which changed in moving to NMCS II. It then examines several of the acceptability issues which were raised by customers, how these issues relate to the technology generation gap, and how they are being addressed going forward. The paper concludes with a discussion of lessons learned, in order to help others avoid falling into the "gap" in the future.

Introduction

Stentor, through Bell Canada and the provincial telephone companies in Canada (Telcos), offer a national fractional T-1 service known as MegastreamTM targeted at companies and institutions which use telecommunications in their mission-critical applications. Megastream is an end-to-end fully integrated digital network solution incorporating hardware, management systems and customer services, which together form a unified solution for a customer's dedicated networking needs.

An important component of the Megastream service is the Network Management and Control System (NMCS), a network management system which allows end-users to control the allocation and routing of their bandwidth to match their business needs. It also provides additional features such as fault management (e.g., testing customer circuit integrity) and disaster recovery capabilities. NMCS is also used by the Telcos to configure and test the physical elements in the network, and provide additional support to the end-users through a unified view of the customer's network.

The first version of NMCS ran on a PC (NMCS-PC). It was first introduced in 1987, and it was a successful product. However it was a victim of its own success, as it was unable to evolve fast enough to meet customer demands for increased capacity (larger networks) and services (new network equipment and services). One

of the major impediments to evolution was the technology base upon which NMCS-PC was developed:

- a segmented hardware architecture, which required application code constructs to manage memory segments
- QNX operating system, which was not readily portable to other hardware platforms
- a proprietary database system which did not allow easy export of management data
- character-based user interface, which limited the amount of information which could be managed on the display

NMCS II Requirements

In 1991 Stentor Resource Centre, Inc. (SRCI), the development organization for Bell Canada and the Telcos, contracted with Teknekron Communication Systems, Inc. (TCSI) to build a new version of NMCS. NMCS II was designed to address the limitations of NMCS-PC and provide significant new services and features. The requirements for NMCS II stated that the new system should be compliant with many formal and de facto standards, including:

- OSI communication interfaces
- OSI Network Management Forum object classes
- POSIX
- Sun SPARC workstations
- Ingres and SQL
- C or C++
- X11R5 and Motif

The requirements mandated a "modular and scalable hardware/software architecture" using managed object (MO) classes and instances. It was required to support multiple, simultaneous users logged-in from geographically-remote sites. The user interface was to be graphical, and provide a distinctive look-and-feel.

The functional requirements included the definition of 40+ Managed Object classes, where each instance of a class would typically contain 20-40 attributes and define 5-15 operations. These functional requirements were a significant superset of the functionality implemented in NMCS-PC, supporting several new Managed Object classes and behaviors corresponding to new network elements.

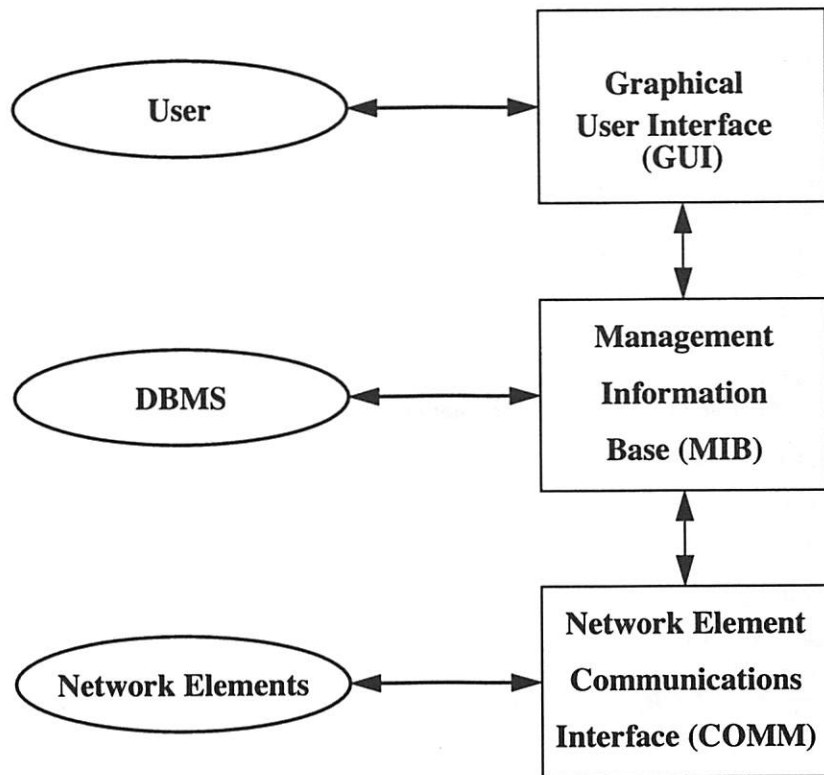
The functional requirements were organized into four categories:

- Configuration Management:
MOs which model the physical and logical elements of the managed network, such as multiplexers, links, and circuits, along with the operations to configure, audit, and test these objects. The components of the managed network are referred to as network elements, or NEs.
- Fault Management:
MOs to represent the alarms and other events which are signalled by the managed network, and logging tools to maintain a record of these events together with an audit trail of user actions.
- System Management:
MOs to represent various aspects of the NMCS II system itself, *e.g.*, the users authorized to use the system and database backup options.
- Utilities:
Tools such as a calculator, calendar, notes, and UNIX windows.

NMCS II Architecture

NMCS II is based upon a distributed process architecture, summarized in Figure 1. The functional components, represented by the boxes on the right, are built on top of NMS/Core[®], a distributed, client-server, object oriented development platform developed and licensed by TCSI. The ovals on the left side represent the elements external to NMCS II.

Figure 1: NMCS II System Architecture



Management Information Base. The heart of the NMCS II system is a Management Information Base (MIB), an object oriented repository of all Managed Object instances in a customer installation. The MIB is responsible for concurrency management (for multi-user operation), validation of all changes to MOs, and for implementing the main functions of the system: monitoring and configuration of the Network Elements (NEs). The MIB uses a relational database (Ingres) for persistent storage of its objects.

The MIB is implemented by two distinct programs. The Master MIB (MMIB) is a single process which is the "owner" of all managed objects. It is the only process which can update the database, and provides a single point to manage concurrency and consistency. The MMIB implements a "lazy" object locking mechanism which allows a form on the GUI to be modeless, *i.e.*, a user can modify any object without requesting an object lock beforehand.

The Slave MIB (SMIB) program functions as a "write-through cache," providing high performance access to view MOs without loading-down the single MMIB server. The SMIB reads and caches MOs from the database on start-up, and can then respond to any read request from the GUI from its local cache. Any attempt to modify an object passes through the SMIB to the MMIB, which validates the request before updating the database. Changes in the MMIB are communicated by events to all SMIBs, which update their cached data accordingly. Each NMCS II installation can support an unlimited number of SMIB processes.

The MMIB and SMIB use the same internal architecture and share much of their base code, which minimizes their development cost.

Graphical User Interface. The Graphical User Interface (GUI) communicates with the users of NMCS II through a set of forms (windows) which are displayed on the user's workstation. The GUI is an X11 client, providing a variety of list, detail, and graphical forms to view and modify MOs. The forms are built using OSF/Motif widgets, but implement a look-and-feel quite different from that mandated by the Motif Style Guide. The GUI uses uid files (a Motif standard representation for form/widget layout) to simplify the development process and provide for easy field upgrade of specific forms.

The GUI forwards requests to an SMIB process to fetch and modify MOs, and to invoke operations on MOs. Each NMCS II installation supports an unlimited number of simultaneous users, each communicating through their own GUI process to the MIB.

Network Communications Interface. The MMIB communicates to the NEs through the COMM process. COMM's main responsibility is to drive the serial line connected to the managed network, translating commands and responses between the NMCS II internal format and datagram format used by the NEs. The MIB uses a non-blocking message-based interface to communicate with the COMM, and can process other requests while waiting for a response from an NE. Each NMCS II installation has a single COMM process.

Size and Scope. NMCS II is a large-scale application, and is the result of many person-years of development and testing. The following metrics give a rough feel for the size and scope of the system:

- over 500,000 lines of C++, C, UIL, and embedded SQL code
- over 3,000 C++ classes
- over 7,000 test cases in the system test case library
- over 150 Motif forms
- over 3 years in development.

Process Structure. The MMIB, database, and COMM processes typically reside on one or more server machines located in a central office location, for ease of maintenance by telephone company (Telco) personnel. In contrast, the client workstations can be located in customer sites, and connected to the server machine by a dedicated 56k facility. Each client workstation can contain a GUI and an SMIB process, and there are no fixed limits to the number of client workstations. (See Figure 2).

All processes communicate using a single universal message format, and use XDR for encoding data. The message format is proprietary, and closely matches the format of internal data structures. Distribution and location/binding of processes is done through simple configuration files, and is quite flexible; there is no fixed relationship between processes and processors, and processes can be combined or separated freely.

Technology Generations

To the author's best knowledge, NMCS II is the most sophisticated and feature-laden network management systems for Fractional-T1 networks in commercial use. Delivering a system of this complexity and magnitude was a difficult undertaking, for both the development and deployment teams. The NMCS II system is based upon a major technology base upgrade from that used in NMCS-PC. That technology base includes many components, which are briefly described below.

For each dimension of the technology base, we briefly describe the selected technology and identify some of the major issues we encountered during the development process itself. We will discuss the impact of these technologies on the end-users in the latter half of this paper.

The diagram illustrates a distributed database architecture. It features two client workstations, Client Workstation 1 and Client Workstation 2, and a central Server Machine. Each client workstation contains a User Interface (UI) and a Slave MIB (Master Information Base). The Server Machine contains a DBMS (Database Management System), a Master MIB, and a COMM (Communication) module. The DBMS and Master MIB are connected by a bidirectional arrow. The Master MIB and COMM module are also connected by a bidirectional arrow. Data flows from the DBMS to the Slave MIBs. Events flow from the Master MIB to the Slave MIBs. Queries flow from the Slave MIBs to the DBMS. Updates flow from the Slave MIBs to the Master MIB.

```

graph TD
    subgraph Client_Workstation_1 [Client Workstation 1]
        UI1[UI] <--> SlaveMIB1[Slave MIB]
    end
    subgraph Client_Workstation_2 [Client Workstation 2]
        UI2[UI] <--> SlaveMIB2[Slave MIB]
    end
    subgraph Server_Machine [Server Machine]
        DBMS[DBMS] <--> MasterMIB[Master MIB]
        MasterMIB <--> COMM[COMM]
    end
    DBMS -- Data --> SlaveMIB1
    DBMS -- Data --> SlaveMIB2
    SlaveMIB1 -- Queries --> DBMS
    SlaveMIB2 -- Queries --> DBMS
    SlaveMIB1 -- Updates --> MasterMIB
    SlaveMIB2 -- Updates --> MasterMIB
    MasterMIB -- Events --> SlaveMIB1
    MasterMIB -- Events --> SlaveMIB2

```

Application Architecture. NMCS PC was designed and developed using traditional procedural programming techniques, although the designers had started to think of the application in object oriented terms. This perspective was evident in the functional requirements, which specified managed object classes (the “leaf” classes) but did not use inheritance or other techniques to manage design complexity and promote reuse.

163

Hardware Platform and Operating System. NMCS II is delivered on Sun SPARCStation platforms, compared to an AT-class machine for NMCS-PC. The initial development was performed on SS1+ and SS2 workstations, but these machines have been manufacturing discontinued in Canada. Porting to the SS10 required more work than expected due to configuration differences in the X and Motif libraries.

NMCS-PC was implemented on QNX, a UNIX-like real-time operating system for PCs, while NMCS II is built upon SunOS. We have had relatively few problems with SunOS, but we experienced issues upgrading, from version 4.1.1 to 4.1.3. A port to Solaris 2.x is on the horizon, with unknown but potentially significant upgrade costs.

User Interface. NMCS-PC's user interface was based on character graphics. It was fast and easy to use, but was limited in the amount and complexity of information it could display and the ability to navigate between parts of the system. The NMCS II GUI overcomes these problems through the use of multiple windows, graphics, multiple display formats, and enhanced navigation capabilities which are common to many Motif applications. However, we have experienced many problems with moving to a windows-based interface, some generic and some specific to Motif.

- difficulty in configuring insensitive fonts, keybindings, and other "minor" aesthetic concerns
- memory leaks in Motif 1.1
- difficulty upgrading from Motif 1.1 to 1.2
- difficulty finding commercial vendors for X11 and Motif able to provide complete support services.

Database. NMCS II uses the Ingres relation database, compared to a proprietary, in-core database used in NMCS-PC. This decision has major advantages in the design of the database schemas, producing reports, allowing the database engine to be distributed to high-performance servers, and allowing integration with other Stentor data systems. However, we had to overcome several problems to realize these advantages:

- Ingres does not "officially" support C++, so we had to exercise care in embedding SQL in our C++ source code
- we had to develop our own "object oriented database" layer on top of the relational technology
- relational technology imposes a performance overhead for simple tasks like event logging
- version upgrades of the supported product required time and resources of the development team to install Ingres, recompile code, and port existing data.

Managed Equipment. One of the main functional requirements for NMCS II was to support a new generation of T-1 equipment. This generation included a customer-programmable cross-connect (switch), a low-end multiplexer, new logical objects for managing services, and a variety of new access equipment. The incorporation of these new features into the functional model implemented in NMCS-PC varied in difficulty. Some of the new equipment could be added by simple subclassing from existing superclasses. In the worst case, however, we had to define radically new objects and management algorithms to support user-level abstractions which were not implemented in the managed equipment.

In addition to the new equipment, the manufacturer produced periodic firmware releases for the managed equipment during the development cycle. These releases fixed problems in previous releases and added new features. Upgrading to these releases required downtime in our testing lab, as well as disruptions when new problems were discovered in the new releases.

Acceptance vs. Acceptability

The NMCS II system is a state-of-the-art application utilizing advanced hardware and software technologies and object oriented development methodologies. The delivered product has successfully met its stated

acceptance requirements. However, the final test of the product is acceptance by its users — the Megastream customers and the support organizations which must support the product and the end users.

In evaluating NMCS II it is important to distinguish between the acceptance of the product and the acceptability of the product. *Acceptance* is a objective concept which can be precisely measured, and is defined as the ability of the product to meet the requirements imposed upon it and pass a formal acceptance test. Final Acceptance is a formal contract milestone between the supplier of the product (TCSI) and the customer (SRCI).

In contrast, *acceptability* is a subjective measure of the usefulness of the product to meet the users' requirements. In a perfect world, the users' requirements exactly match the formal product requirements, and acceptance and acceptability are equivalent. In the real world, however, it is almost impossible to precisely capture the users' requirements in a formal sense. This is due to a number of factors:

- Users are network managers or system administrators, not software developers. They understand their job in terms of their tasks and responsibilities, but additional analysis is required to translate those tasks and responsibilities into functional requirements.
- Prototypes are useful but of restricted value. A prototype is useful for a user to evaluate look-and-feel issues and comment on new features, it does not allow her/him to test the product in an operational scenario.
- Due to schedule and budget restrictions, it is not always possible to fully satisfy the users' collective requirements in a first release.

Each of these points can be addressed to some degree using a variety of techniques such as focus groups, early involvement of users in product testing, and proper expectation management in the face of limited resources and phased delivery schedules.

The final challenge to NMCS II acceptability is the existence of NMCS-PC. As described earlier NMCS II was to replace NMCS-PC functionally, as well as provide a new usage paradigm based on the new technology generations in the delivery platform. While the functionality of NMCS II was generally acceptable, the paradigm change created a significant gap in the perceived usability of the product. This "technology generation gap" manifests itself in three key areas:

- Usability
- Performance
- Deployment

The first two areas are of primary concern to the end-user community — the network operators and managers responsible for managing the network to support the business applications which use the network. The third area is of concern to the Telco operators, who are responsible for maintaining the NMCS II application and answering questions from the end-user community. For each of these areas we will describe some of the feedback we have received from the appropriate user community, and how we are addressing the acceptability issues raised by those users.

Usability

Usability can be defined loosely as the ease with which the user can do his job using NMCS II. Some early users were unhappy with the usability of the first release of the product; many of their complaints can be traced directly to the technology generation gap.

User Interface. NMCS-PC used a character-base user interface. It presented a limited amount of information on the screen at one time, and relied heavily on function keys to navigate between screens and invoke actions. NMCS II, in contrast, makes extensive use of windows and the mouse.

As we delivered the first release of NMCS II we expected a transition period during which users would have to familiarize themselves with the mouse-and-windows interface. Many of the problems the users reported indicated that the transition is going to require more time than we had anticipated; some of the problems or enhancement requests we heard from users included:

- there is too much information on the display because of the ability to open multiple windows
- data entry and operation invocation require the mouse; many users dislike the mouse and prefer function keys
- users prefer to operate from a tabular list of objects, rather than a detailed view of individual objects; this reflects the list-oriented behavior of NMCS-PC.

Taken as a whole, these concerns indicate that the technology change from character to graphical user interfaces can greatly impact the usability of a product, particularly if the users do not have prior exposure to graphical user interfaces. In follow-on release we have been adding features to NMCS II to make it less dependent on the mouse, and make it easier to work from lists of objects.

Functionality. Some of the usability problems users reported were due to changes in the way functionality was designed and implemented between the two products. We have received comments of the form "NMCS II does XYZ differently than NMCS-PC." This reflects a natural inertia on the part of the users, who have become proficient with their current system and are reluctant to change the way they work.

Many of the changes to existing functionality were required to support features or functions introduced in NMCS II, and it would have been impossible to support both modes of operation in NMCS II. We are addressing this issue in future releases by involving users early in the testing cycle at TCSI offices in Berkeley. This approach provides a number of benefits:

- Users have an opportunity to test new features in the context of an operational product, not a prototype. If there are usability issues identified the development team can take steps to correct the problem before the product is delivered.
- Software developers and testers have an opportunity to learn how users work with the product in the field, which can increase our internal testing effectiveness.
- By socializing with the development team users become "part of the team," which opens the communication channels and provides a rich source of requirements and new ideas for follow-on development.

Performance

We define performance as the time taken to complete a given operation, and includes not only the performance of the application software, database system, and networks, but also the time taken by the user as part of the operation. Many of the performance issues which have gated product acceptability are due to the technology generation gap.

User Interface. Changing from a character-based user interface to a full X/Motif windowing interface has resulted in a noticeable degradation in responsiveness. The amount of tuning which can be achieved is limited by the complexity of the individual screens and the overhead of accessing Motif uid files.

Database. NMCS-PC utilized a custom database which was tuned for the application, but did not readily support queries from external processes. Switching to a commercial relational database in NCMS II allowed us to rely on the standard database capabilities like backup and recovery, to "open up" the data stored in the application to other applications, and to enable the use of SQL-based tools for custom reports and other queries. These capabilities come at the cost of performance, however, as even a highly tuned commercial RDBMS cannot compete with tuned in-core structures.

Functionality. There are several cases in which functions or features introduced in NMCS II have significant negative impacts on a user's productivity. For instance, a major change in functionality from NCMS-PC to NMCS-II was the addition of element management capabilities for the multiplexers. This new function requires the creation of several hundred individual objects for each multiplexer. Although the time to save an object to the MIB is less than 5 seconds, the time to open the screen and enter the configuration data for each object is over 1 minute.

Performance Improvement. Most performance acceptability issues can be traced to comparisons to the NMCS-PC product. NMCS-PC is a mature application, it has been highly tuned over its lifetime, and users have adapted to the product and have found the most efficient ways to use it. The performance degradation inherent in the technology changes described above, coupled with the high performance standard set by NMCS-PC, virtually guaranteed that users would rate NMCS II slower than NMCS-PC.

We have addressed the performance concern in two key ways:

- Continued incremental performance improvements in all areas of the system, including software optimization, database tuning, hardware upgrades, and reconfiguration of the client-server environment to maximize throughput.
- Redesigning the tasks which operators perform with NMCS II to reduce the operator interaction. For instance, in a pending release we will provide a capability to partially provision a database by uploading information automatically from the network. While this operation takes considerably more time than creating an individual object, it can replace up to 150 individual operations, resulting in two orders-of-magnitude improvement in user productivity.

Recent efforts in these two areas have increased the overall system performance to acceptable levels, and we have committed to a program of continuous performance improvement throughout the NMCS II product lifecycle.

Deployment

Deployment issues include Operations, Administration and Maintenance, and are the primary concern of the Telco users. As with the other two areas — Usability and Performance — many of the acceptability issues arise from the technology generation gap.

Hardware Platform and Operating System. NMCS-PC, as its name implies, was delivered on an Intel-based personal computer running QNX, and it used VT100-compatible terminals over dial-up lines for remote access. In contrast, NMCS II is delivered on Sun SPARCStation servers, and uses SPARCStation workstations over LAN or dedicated links for remote access; both client and server platforms run the SunOS operating system. In our experience, and in the feedback we have received from the Telco users, maintaining networked UNIX workstations is much more difficult than the equivalent number of PCs, as it requires more staff with greater knowledge and experience. Many of the problems which the users have experienced in this area are part of the steep learning curve which is associated with converting to UNIX.

Application Architecture. NMCS II uses a distributed process and data architecture, with the SMIB/MMIB separation as a key architectural component. Each executable utilizes a number of libraries, utility programs, and configuration files to ensure proper operation. The GUI process requires several hundred auxiliary files which contain the screen definitions (Motif uid files), online help, and other configuration data. As a whole, an NMCS II distribution tape includes over 1500 files in 40 directories.

For customer installations, the basic software components are configured on top of a LAN/WAN environment which provides the distribution infrastructure. In contrast to the dial-up VT100 access of NMCS-PC, the dedicated 56k facilities used by NMCS II require additional hardware (routers and bridges) and expertise to configure and maintain.

We spent considerable time developing maintenance and packaging scripts to configure an application, based on a "reference configuration" targeting a LAN environment. This configuration did not always match the particular needs of individual customers, particularly those customers with extensive WAN requirements. As a consequence, the operations staff has had to invest considerable time producing and maintaining custom configurations.

Database. The Ingres relational database which NMCS II uses is itself a complex product, which requires specialized knowledge to install and maintain.

Deployment Improvement. Most of the issues listed above arise from the additional complexities introduced with the technology change. We are approaching these issues with a multi-prong approach.

- Increase the level of expertise within the operations staff, through training (formal and on-the-job) and consultants.
- Formally address the requirements for supporting custom product configurations, and deliver enhanced installation and maintenance scripts with new releases.
- Simplify the operation, packaging and installation.

Lessons Learned

The NMCS II development and deployment effort has been a challenging and rewarding undertaking. Both SRCI and TCSI have learned some lessons as we have bridged the technology generation gap, which we summarize below.

Comparison to the "Old System". Whenever a new system is being built which replaces an existing system, the new system will always be evaluated against the performance and usability of the system it replaces. While this comparison may seem unfair to developers, especially if the new system provides greater functionality or implements a different way of performing current tasks, the subjective impressions resulting from this comparison become real issues which must be addressed.

In some cases these issues may be antithetical to the basic design of the new system; education of the users on the design and usage of the product are the best solution to overcome these issues. In other cases, compatibility modes can ease the transition to the new system. In either case, the requirements analysts of the new system should fully understand how users use the old system — what are their likes and dislikes — and plan to explicitly address those features in the design of the new system.

End User Involvement. We cannot overestimate the importance and value of involving the end users in all stages of the product development life-cycle. Typically users are consulted during the requirements phase, and again during the acceptance phase. We have found that keeping users involved through the development and early testing phases improves the quality of the delivered product, opens the communication channels between the development and user communities, and can address and resolve usability issues before they become problems.

End user involvement would have been particularly beneficial in evaluating the new usage paradigm as implemented in the Graphical User Interface. The change from text to graphical presentations, with the corresponding change from function keys to the mouse input, is the first hurdle users will encounter with the NMCS II product. Exposure to prototypes and early releases could have smoothed that transition. In addition, the choice of uid files to define screen layouts allows field personnel to modify screens to conform to user requirements, without requiring changes to application code.

Attention to Deployment. Products like NMCS II, which have a relatively small number of installed sites but are custom-configured for particular customer needs, require special attention to deployment issues.

Focusing on packaging and installing a “reference configuration” is not adequate under these circumstances. That is, the requirements analysis and definition process should include the needs of custom configurations. Stated another way, the deployment staff are part of the user community and must be included in normal product definition and development life-cycle.

Expectation Management and Continuous Improvement. In retrospect, the goal of delivering the NMCS II product, with all existing NMCS-PC and new functionality, on the new technology base, with the same performance and usability as NMCS-PC, *in the first release*, was over-optimistic. A product of this complexity requires one or more iterations to work the kinks out. We have adopted a program of continuous improvements to NMCS II through the maintenance and upgrade program. Coupled with tighter management of customer expectations — and greater visibility of customers into the improvement process — have greatly increased customer satisfaction with the product.

As part of the process of continuous improvement, we have taken steps to shorten the development cycle for follow-on releases. Reducing the time between releases allows us to react quicker to user issues as they arise, and makes very visible to the user the improvements. We are shortening the development cycle through a combination of the following techniques:

- Reducing the feature content of incremental releases, deferring items of lesser importance to later releases.
- Including members of the development team part in the requirements analysis process. The developer is best able to propose simple functional alternatives which satisfy user requirements.
- Beginning requirements analysis in parallel with the development cycle. This pipelined approach allows us to further reduce the time between releases without squeezing the development cycle.

Conclusion

Upgrading an existing application to a new technology base has many inherent challenges above and beyond the software development *per se*. The technology generation gap can introduce acceptability issues for a product which may not be captured by the formal acceptance process, in areas like usability, performance, and deployment. The best solution to bridging the gap is to recognize its existence in the first place, determine how the gap will impact the user community, and engage the users in the solution process.

Acknowledgments

I would like to thank the entire NMCS II development team in SRCI and TCSI, particularly Steve Gale, Pierre Osborne, John Reid, Denise Semak, and Pierre Veilleux (SRCI) and Ira Rotenberg (TCSI), for delivery a quality product in which we can all take pride. Collectively we owe thanks to our user community for their patience and feedback, especially Jerry Abrams, Luc Bourdages, Kevin Johnson, Tracy Kluczynski, Eloi Leroux, and Bruno Smith. Final thanks go to Ram Banin, Dennis Boyko, Marc Farmer, and Prasad Yerneni for reviewing early drafts of this paper.

References and Additional Information

In early 1993 AT&T Network Systems and TCSI agreed to integrate and jointly market the NMS/Core product and BaseWorXTM application platform, with the NMS/Core forming the Object Services Package (OSP) of the BaseWorX platform. For additional information on BaseWorX/OSP please contact the author.

Megastream is a trademark of Bell Canada. NMS/Core is a registered trademark of Teknekron Communications Systems, Inc. BaseWorX is a trademark of AT&T Network Systems.

Porting and Maintaining with X and Motif A Retrospective View

Paul Davey

User Interface Technologies Ltd.
17-21 Sturton Street
Cambridge
CB1 2SN
England
email: pd@uit.co.uk

ABSTRACT

In 1989 I accepted a job with IXI Ltd. in Cambridge, England, a small software house specializing in the X Window System. The flagship product, X.desktop, was then moving into its third major version (2.0), the first based on Motif.

Initially IXI was establishing X.desktop, and the concept of desktop software for Unix and X, but as time went the initial issues of rapid portability and flexibility, were replaced by other factors as the company, market and product all grew larger and more complex.

Maturing code, increasing numbers of programmers and staff turnover all played their part in making the management of the ongoing project more and more challenging. When overseeing the porting and maintenance of X.desktop in 1992, I found myself dealing concurrently with three major versions (2.0, 3.0 and 3.5), existing on over 30 platforms (with more than 50 active configurations).

After successfully streamlining the porting and maintenance process I was then asked to organize the Motif toolkit team along similar lines in August 1993. This was subtly different from the X.desktop work since the code originated externally (from the Open Software Foundation), and had to meet open as opposed to proprietary standards.

Keen competition in the market providing Motif development libraries on Sun meant that high quality and ease of use was highly important. As with X.desktop a rapid response to customer problems was another important factor in keeping customers satisfied.

This talk will examine some of the lessons learned about the development of multi-platform applications and toolkits over the software life cycle of X.desktop. Some additional examples are taken from the work porting and enhancing the Motif toolkit.

Although a premier X application X.desktop code was only partly X based, much of it being either Unix system related or just plain data manipulation. The uses (and abuses) of standard Unix software tools such as yacc, lex and SCCS will be illustrated. The parallel development of a problem and bug tracking systems will be described as it resulted in the customized internal tool in use at the end of 1993.

Initial Development

The very first version of X.desktop was a pure Xlib client (Xt was still under basic development at that time), but this only truly existed as a prototype to illustrate the potential of X and Unix with a Macintosh like face. The first version seen by end users (version 1.3) was Athena widget based but retained the ability to build the pure Xlib version for platforms without an Xt and Athena widget port. The cpp conditional macros supporting this remained in the product source code throughout the first Motif toolkit version (2.0).

Lesson one is that your prototypes often end up as the basis for products. Therefore they should be as carefully written and documented as time and circumstance allow. Cpp macros can be used to select different modules. I estimate that 10 to 20% of the X.desktop 2.0 code was un-compiled (and un-compilable), but was too intermixed with the generic and Intrinsics and Motif widget set code to be removed safely in a reasonable time frame.

Designing For Portability

In order to establish X.desktop as the premier desktop for X it had to be easily portable. This was one of IXI's major successes, the quality of the code and its structure being essential to this.

Two major factors contributed to this. Firstly the strength of the X Window System standards, which meant that APIs were clearly defined, particularly so in comparison to Unix. Secondly the use of a virtual interface to the OS where necessary to hide non standard implementations. Known as OSAL (Operating Systems Abstraction Layer) the interface emulated a subset of POSIX, by overloading system or library functions with wrapper functions named transparently by defining and redefining in header files. All usual source code changes (with one exception) were made in just two header files. One setting environment variables for the build processes, the other setting tokens in a per system header file.

This approach worked very well, and so long as programmers were au fait with non standard functions - this knowledge being enhanced by developing in parallel on as many machines as possible standard Unix code (ie somewhere between POSIX, BSD and System V) - porting issues were typically where are the X libraries on this system, or what is the Motif Intrinsics library called today. The most notable failure was to an parallel Unix like machine (which had best remain anonymous), which failed to implement a fork system call properly. This proved somewhat of a handicap on an application designed to launch other tasks.

The nature of X Window System software where applications have binaries and associated resource database files and of a program as configurable as X.desktop (which can be thought of as a graphical shell with more features and predefined functions than say the Korn shell or BASH) meant that after compilation some other configuration had to be met.

With all the main Unix variants then being either AT&T or BSD based this should not have been a problem but at least one port tried to display an icon for the executable file /usr/ucb/man although on that particular platform the /usr/ucb directory did not exist at all and the man command was located in /usr/bin. In retrospect a virtual command layer would have been a boon, a Shell Command Abstraction Layer if you will.

Lesson: Use virtual environments to hide all system differences

Install Scripts

There was also the problem of installing and un-installing the software on client systems. A customer unloaded a tape then ran a script which installed the binaries and other files on the system. This worked without any major incidents, although the un-install script never quite made it to the point of working until a new engineer interpreted the instructions for writing it slightly differently. "Tar the directory onto the tape" was locally interpreted as

```
tar c directory
```

but the new employee gave the commands

```
cd directory
tar c .
```

This did not cause the install procedure to fail, rather than containing all the required files in a subdirectory they were unloaded in the current directory of the customer installing it. The file INSTALL still existed and when run it worked perfectly. Too perfectly since its final line cleaning up was `rm -rf *`. As Murphy's Law would have it the customer receiving the tape unloaded it the root directory and lost an entire system.

Lesson: Be very wary of wild cards and rm statements in scripts.

Install scripts seem to gain a life of their own. There is always a customer who wants to locate some part of a product a little differently, a previous version of the install script for IXI Motif had become the source of

the majority of support calls as it allowed users to place parts of the distribution in half a dozen locations.

Un-installing into a single tree means that it is easy to check for sufficient disc space, and this can be made to be the installing user's or system administrator's responsibility. If installation fails at the first attempt, or if several related products are to be unloaded, then install trees can be overlaid without problems, and it is easy to delete the software manually.

For convenience files that that might be placed in standard locations (for example /usr/lib or /usr/bin/X11) can be placed elsewhere and then soft-links made manually or automatically from standard locations to files and directories in the application tree.

Even if speed issues mandate some site dependent configuration this should be kept to a minimum and done dynamically at runtime avoiding any dependencies on installed paths. Configuration should be allowed rather than required. Absolute paths are fundamentally inflexible and should be avoided wherever possible. Setting (or deducing) a single environment variable indicating the top of an installed software hierarchy should allow enough flexibility for all file locations. These rules will allow software to be run from any tree, including directly from CDROM.

Lesson: Keep distributions to a single tree wherever you can and if possible avoid any post install processing.

Using Generated Code

Where software tools are used which produce automatic code the temptation to edit it by hand should be avoided. X.desktop 2.0 had yacc output that was tweaked for efficiency and could not be regenerated automatically. Only a parser rewrite in version 3.0 solved the problem. If generated code must be altered, it should be clearly documented or automated with a tool such as sed or awk.

The use of such tools also requires some care. I have been amazed on many occasions by the lack of knowledge of highly skilled C programmers of the basic variations in user commands and system facilities between BSD and System V based systems.

Lesson: If you intend to port to these machines and to use the shell and utilities then the same care must be taken as when using system library code. A vendors added value can easily become subtracted value when porting to other platforms.

The COSE initiative and CDE project will surely aid these issues, but due to legacy systems they will not be on every target machine for several years to come.

Version Control

Obviously version control is a must for all serious projects. The choice of systems seems to be threefold in type, viz a standard provided utility, ie normally SCCS, generic public domain utilities (for example RCS or CVS), or complex software engineering tools such as Code Center. While the latter types of tool can be very expensive straight SCCS seems to me to be rather primitive (as it is over 20 years old) and requires a growing number of ever more complex scripts to adapt it to changing requirements.

Other companies or organizations will always use other systems. For instance the X Consortium uses RCS, while system vendors may use their own proprietary tools or plain or enhanced SCCS.

Ideally not a single line of source should ever be duplicated, and functions should be placed in internal libraries so they can more easily be shared between major versions and different products.

In the real world though there is no ideal solution. Projects grow out of early prototypes adapted from other code.

Lesson: Use the best tools you can get or afford. Plan time for reviewing and if necessary replacing or updating internal tools.

Since programmers like to code there is a danger that code will be duplicated rather than systems reorganized to minimize maintenance. One project I worked on had three separate trees based on the same core code. Needless to say they evolved to be incompatible with each other and bug fixing was scaled up by a factor of three.

Lesson: Key functions should be placed in internal libraries or modules and extracted into build directories when needed. Time spent extracting generic functionality from project specific modules will be paid back by a factor of 10 or more later in the software life cycle.

Internal Support and Development Tree Hierarchies

Sharing resources such as disc space and CPU time can cause friction when deadlines and tempers are short. Programmers are the worst people to organize their code in large projects. They are often too wrapped up in the small scale details to take larger view. Good internal support saves a great deal of expensive development time and can also provide a project neutral overview of facilities and procedures.

When maintaining large projects a clear network layout will save you a great deal of time. SCCS and source trees are simple enough, but when the source tree contains a distrib tree, (with all the configuration and binary files to be shipped) and that distrib tree is extracted from an SCCS distrib tree at build time then some clarification is needed. For X.desktop we used the following terms for trees:

- SCCS- SCCS s. and p. files
- OUT - Source code versions extracted from the SCCS tree
- BUILD - Trees for building a port in,
These had source files linked into the OUT tree so
that source files could be shared between ports.
- SHIP - Tree's containing final distributions, as delivered
to customers.

Tree's named distrib and source were deliberately not used as their names were already associated with the older system.

Procedures were set in place for updating the latest OUT tree only when a fix was stable and the use of multiple OUT tree's allowed a form of product version control. For example, many ports could be built from the 3.0C8-OUT tree, keeping only one copy of the sources on line, reducing not only disc space but also the chances of unintentional changes to the source code.

Automated Building:

Imake, the X Window System meta-makefile tool was quite rough and ready when X.desktop was first conceived. To allow easier porting X.desktop used its own simple set of scripts and regular Makefiles, with (as I remember it) only one problem relating to the inclusion of other Makefiles. Nevertheless as more systems were ported to some extra issues evolved. Most notably amount of manual intervention required became an issue. While it was felt that imake was becoming standard enough to be useful the investment of time to rework the build system to use it was too great. However the shell based system proved to take longer to implement than the developer had predicted, and with hindsight it would have been better to bite the bullet and develop Imake rules for the many tables in X.desktop.

Motif does use the imake system and integrates well with X, however the OSF developers clearly built on machines where either the X Window System was installed in the usual MIT locations (/usr/lib/X11, /usr/bin/X11 and /usr/include/X11), and/or the build trees for X and Motif were in the same hierarchy. This proved to be hard, but not impossible to resolve with the Sun OpenWindows hierarchy under /usr/openwin, and configuration files allowing a (theoretically) hands off build from a makefile now ship with IXI's Motif development kit.

Lesson: Never trust a developer's schedules. What seems to be a simple task can be found to be more and more complex as work progresses. Using appropriate tools even if you are initially unfamiliar with them can save time at a later stage. It is worth being prepared to support some tools on development platforms lacking them.

Testing:

Testing is of course a vital part of any release cycle, but restrictions of time and the lack of automated test systems for X until recently made X.desktop time consuming to test. What could be done was done. The OSAL implementation had its own tests run as apart of the build procedure, which would diagnose any

non-standard system functions and external APIs could be driven by test programs designed to elicit expected responses. GUI based testing was completely manual and was only applied in full to major releases. Basic functionality was tested for each port, initially by the porting engineer, but in time a separate QA department was established. Many problems turned out to be X server dependent but despite good intentions, repeating manual testing 10 or more times with different servers was not practical.

Testing Motif toolkit ports was somewhat more straight forward as the OSF kindly provided a manual test suite at first and later a fully automated verification test suite. The X Test Suite will make a major contribution to improving the implemented standards of the X Window System, making porting engineers' work that little bit easier.

Lesson: Like the rebuilding of products, software testing is needed repeatedly when working with a large and mature code base. One person's fix is another person's bug. Automated testing provides the best measure of software acceptability.

Enhancements

Many features have been added to X.desktop over its lifespan. Although the original product architect was involved with the majority of these new features, many other personnel carried out design and implementation. For X.desktop as a proprietary product the main issues when adding functionality were, a strong commitment to backwards compatibility, and change control provided by a separate specification. (The issue of how changes were made to that specification is not one that can be covered here.)

For the Motif toolkit enhancements took several forms. Additional demonstration and sample code is a useful, and portable added value but three types of changes were made to the basic Motif source code. These were transparent changes, where a feature might be re-implemented for greater efficiency or for additional fallbacks; translucent changes, which altered the behavior of the standard distribution to match the specification; and added value, where additional features are provided.

Transparent changes have few if any implications for the developer user, but added value needs controlling to avoid accidental use leading to portability problems. As in IXI Motif this can be provided by cpp macros at compilation time, or environment variables at runtime, allowing or disallowing some or all of the extended features. Translucent changes should not need any control, but with development tools used as widely as Motif and X, bugs in the standard distributions become expected behavior, and therefore a de facto standard which should be supported with control as for added value.

Bug Tracking

The original problem tracking system used at IXI was shell script and file system based. Slow, and without file locking, it had many disadvantages but on its positive side side were flexibility and cost effectiveness. In time as the volume of support work rose rapidly, a search was made for a more sophisticated system, but no suitable product could be found available off the shelf and so a back end database system with a customization facility (in Prolog), and a Motif based GUI was purchased.

This meant that the format of the databases making up the system still had to be defined, and much discussion from the support, QA and development departments produced the first iteration of the actual system design. With a little trial and error a workable system was produced, and after a few months some minor changes were made.

The major design features were that while records are held separately for: problems (reported by customers, and dealt with primarily by the support department); bugs and feature requests, (created by the development team in response to testing or customer problems); and releases (controlled by the QA team); these records sets are all interlinked. Different focuses displayed various subsets of records, with predefined focuses existing for typical use by various teams.

A useful bonus to the universal database structure was that management reports can be generated allowing sensible scheduling of work. In bug fixing team members were encouraged to select their work from the overviews of work outstanding, allowing some personal control over a fairly routine task. The major issues with the system from the users point of view were, the initial learning curve required, the fairly basic GUI (how do you make a database as exciting looking as a desktop full of icons?), and the speed of interactive response.

The IBIS system did prove a valuable resource in planning and distributing work between teams and team members. In my own opinion its largest drawback was the lack of integration into the source code and version control systems. Cutting and pasting SCCS identifiers when you have hundreds of modules and versions is hard to enforce - and of course as a computer scientist I expect the machines to do the tedious work for me.

Software Design for Installability

Steve Simmons

Inland Sea

Abstract

In classic software design, little or no consideration is given to the issue of installing the resulting package on the customer machine(s). This is complicated by the rich variety of administrative policies and styles used in UNIX¹ installations. This paper will not attempt to prescribe a specific method for software installation. Instead, it will focus on (a) issues which make a package installable into highly customized sites such that the package can be installed with minimal disruption to both the site and the package and (b) the reprogramming of system configurations in a style designed to minimize the impact of that reprogramming.

Introduction

At first glance, the issues of software quality and software installability are almost completely distinct. There are high-quality software products which are trivial to install, and high-quality software products which are a nightmare. There are small, off-the-cuff programs on the net which are trivially easy to install, and others which give ones nightmare.

After fifteen years as both software developer and system administrator, I have found a number of features of software design which affect the installability of the end product. These are not large features, and during software development they usually seem to be trivial. This paper will attempt to point out what makes software installable and why one particular choice might be superior to another. It will also describe some of the system checks and precautions which a good install script will do.

There are standards efforts under way for software installation [Archer93], but these focus much more on the mechanics of installation rather than the qualities which make a package installable. The resulting standard will be an improvement over the current wild mishmash, but it will not in and of itself improve the quality of the installations. It will simply make them more predictable.

As a part of this, we will occasionally discuss a theoretical word processing package **WhizzyWimp**². All of WhizzyWimps installation features will be based on features of existing software packages. Some of these are taken from good examples found in the real world, others are taken from fixes needed to install recalcitrant packages. The guilty will remain nameless.

Much of this paper will look at the issues of what makes for good installability. Once this is complete, we will turn to how those issues affect the development of the program itself.

What Is Installability

To be installable, a package must meet many criteria. Some are related directly to installation, while others cover the reconfiguration and use of the package after installation.

During the installation process, a well-designed package

¹ UNIX is a trademark of X/Open.

²What You See Is What I Mostly Programmed

- respects the existing technical and political policies of the the site;
- provides installation methods appropriate for both the naive end user and the sophisticated large-site manager;
- is resilient in the face of partial installation (that is, a failed installation can be recognized as such and backed out);
- makes minimal impact on the configuration of the machines on which it runs;
- does not attempt to enforce change of style or environment on the end users or systems managers;
- is resilient in the face of system customization;
- provides a clear indication of what has been installed and where;
- respects file permissions;
- can be upgraded in place without affecting the previous installation;
- permits the simultaneous installation and use of multiple versions;
- gives good error messages in the case of both internal and external failure;
- respects the security of the installation;
- provides good documentation for the system manager.

After installation, a good package

- is uninstallable;
- respects the security of the installation;
- provides good documentation for the system manager.

In summary, a good package has minimum impact on the rest of the system when installed. In actual practice, the total changes can be reduced to the amount of disk space consumed and the appearance of a single file in the location of a sites standard executables.

These are not cast in stone; there may be reasons to violate one or another of the above that are package-related. For example, numerous word processors have one-way conversion of old file formats to new formats. This should not, however, mean that users of the old form must upgrade if there is no need for sharing files.

The benefits increased installability are many. The most important for the supplier are:

- Ease of installation makes "test runs" and product samples much more acceptable. If a product is difficult to install, the likelihood of the installation completing successfully and the customer actually trying out the product are increased.
- Respect for the site (policies, uninstall, resilience) means reduced cost of ownership. This increases the value of the software to the end user and makes future sales more likely.
- Easier installation means greater likeliness of correct installation, reducing the cost of support.
- Many of the features which make a piece of software easier to install also make it easier to verify correctness at a later date; further reducing the cost of support.

In addition, the professional programmer is concerned with the quality of *all* work, not simply the source code. When building a package, the installation and cost of ownership are as important an issue as correct performance. The professional should be as concerned about these

items as about user interface or any other developmental issue.³

Localizing Changes With A Shell Script

By making all system changes in a single area, one can minimize the effect of a change on a given system. This has several additional benefits, which we will discuss as well.

The ideal package consists of a single publicly installed executable which references an area owned only by that package. In that way the system is minimally impacted by the additional software, and the chance of packages interfering with each other is reduced to almost zero.

With WhizzyWimp, it turns out we have a package which consists of a number of separate executables. There is WhizzyWimp itself, the spelling checker, the PostScript⁴ translation package, the index generator, etc, etc. All of these need to be installed in some area where WhizzyWimp can find them, but we want to avoid installing them in the public areas.

What we do is make WhizzyWimp itself begin with a shell script which references a configuration file. This particular model is based on the C News install [Collyer87], but it may well pre-date such use. The WhizzyWimp script is shown in Figure 1. Note that 90% of this file is error checking. Nothing is ever used before its existence is checked, and decent error messages are produced. It's a reasonable example of defensive shell script writing, with an overriding effort to describe the error condition rather than simply letting (non-)execution take its course.

Note the copious use of double quotes and curly braces. The double quotes in the `if` tests insure that uninitialized variables (whether from oversight or typographical errors) will not cause the test to abort. The curly brackets are defensive programming so that future modifications are less likely to affect the resilience of the script. The double quotes in the bodies of the error messages will capture errors from uninitialized variables and from variables which contain spaces or other whitespace.

There are other methods which can be used to test definition of shell variables. In the C News configuration we see constructs like

```
NEWSCTL=${NEWSCTL-/usr/lib/news}
```

While this is both correct and portable, it requires the reader to know shell variable construction rules fairly intimately. The `if` construction is immediately obvious to anyone with the slightest experience in programming, and hence preferable.

The most critical portion is the line

```
${WHIZZYWIMPCONFIG}
```

This is the invocation of the configuration script itself. By using the `.` directive, we can make changes in the local shell script environment. As we add more and more programs to the WhizzyWimp package, each of them refers to this configure script rather than embedding the definitions individually in the programs. When a system change is needed, a single change to the configuration file updates all programs immediately.

The configuration file shown in Figure 2 follows similar paranoid principles. Note that it also error checks for the existence of every file and produces not simply an error message about what is missing, but a message about whether it is the standard or a locally defined alternate which is missing.

The configuration file is also executable on its own, so it may be tested without having to invoke the actual application.

³And packages which are easy to install and maintain often result in gifts of beer at conferences such as this.

⁴PostScript is a trademark of Adobe, Inc.

```

#!/bin/sh
DEFAULT_WW=/usr/local/lib/WhizzyWimp/Version3.1/ww.config
if [ "${WHIZZYWIMPCONFIG}" != "" ] ; then
    ERRMSG="Your sites locally defined"
else
    WHIZZYWIMPCONFIG=${DEFAULT_WW}
    ERRMSG="The standard"
fi
if [ ! -r "${WHIZZYWIMPCONFIG}" ] ; then
    cat << EOF
$ERRMSG master definition file for WhizzyWimp,
"${WHIZZYWIMPCONFIG}", seems to be missing.

Please let your system manager know. WhizzyWimp
cannot run until the file is available.
EOF
    exit
fi
WW_EXEC=${WHIZZYWIMP_HOME}/ww
if [ -x "${WW_EXEC}" ] ; then
    exec "${WW_EXEC}" $@
else
    cat << EOF
The WhizzyWimp executable, "${WW_EXEC}", seems to
be missing. Please let your system manager know.
WhizzyWimp cannot run until the file is available.
EOF
fi

```

Figure 1

```

#!/bin/sh
DEFAULT_WHIZZYWIMP_TOP=/usr/local/lib/WhizzyWimp
if [ "${WHIZZYWIMP_TOP}" = "" ] ; then
    WHIZZYWIMP_TOP="${DEFAULT_WHIZZYWIMP_TOP}"
    ERR_MSG="The default"
else
    ERR_MSG="Your locally defined"
fi
if [ -d "${WHIZZYWIMP_TOP}" ] ; then
    :
else
    cat << EOF
$ERR_MSG master directory for WhizzyWimp, "${WHIZZYWIMP_TOP}",
cannot be found. WhizzyWimp cannot run without that directory
and its contents. Please inform your system manager.
EOF
    exit
fi
DEFAULT_WHIZZYWIMP_VERSION="Version3.3"
if [ "${WHIZZYWIMP_VERSION}" = "" ] ; then
    WHIZZYWIMP_VERSION="${DEFAULT_WHIZZYWIMP_VERSION}"
fi
WHIZZYWIMP_HOME="${WHIZZYWIMP_TOP}/${WHIZZYWIMP_VERSION}"
if [ -d "${WHIZZYWIMP_HOME}" ] ; then
    :
else
    cat << EOF
$ERR_MSG master directory for WhizzyWimp ${WHIZZYWIMP_VERSION},
"${WHIZZYWIMP_HOME}", cannot be found.
WhizzyWimp cannot run without that directory and its contents.
Please inform your system manager.
EOF
    exit
fi

```

Figure 2

All subsidiary programs and files which WhizzyWimp requires should be installed in `${WHIZZYWIMP_HOME}`. The structure under this tree can be as complex as desired. As long as all shell scripts begin by invoking the `${WHIZZYWIMP_CONFIG}` file and all child processes are started from there, the tree can be placed anywhere. Any time system installation requires moving the tree, the system manager can do so without fear of breaking WhizzyWimp.

Since the invocation of WhizzyWimp manipulates the environment, it might be tempting to modify the users `PATH` in the configuration file. Don't! This can result in ugly and subtle errors. WhizzyWimp executables, both the master and any slaves, should reference the environment variables to build paths for any subsidiary executable they invoke. This will avoid problems of name collision.

This method has the additional benefit of allowing installation of multiple versions of WhizzyWimp. Each version gets its own tree under the master WhizzyWimp tree. Individual users can simply put the appropriate version number in their environment, and the right thing will happen. This allows the system manager to install a new version of WhizzyWimp without

disturbing old versions. At some future date, the manager decides the new version is stable and deletes the old version or makes the directory inaccessible. Users of the old version will get a useful error message when this occurs. This also simplifies uninstallability, which we'll return to later.

We have reduced the number of changes visible to the average user to one – the installation of the master WhizzyWimp script in some (any!) normally searched public area. We have not had to modify a users PATH or .cshrc or .login files – always a risky proposition anyway, as we have no reliable way of predicting what shell a user uses or the method by which it is invoked on a given system.

This method permits the sophisticated administrator to take very direct control of the installation location of WhizzyWimp. The default installation method will use the default directories as indicated, the expert method will permit the local administrator to make major changes without affecting the basic run style of WhizzyWimp.

One complicating factor of this method is that it requires the install script build the shell script on the fly. This is a long-solved problem. The reader is referred to the C-News or INN source [Collyer87,Salz92], or any Cygnus install package for examples. Two tools of particular note are the Cygnus configuration utility [Pixley92,Pixley93,Cygnus93] and Larry Wall's Meta-Config [Wall].

When using shell scripts for configuration, one wants to avoid needless repetition of configuration. Since WhizzyWimp has a stand alone spelling utility, we would like a reliable method of determining if the configuration has already been done.

The simplest method is to create an environment variable which is used exclusively for that purpose and check it at the beginning of the configuration file or before doing the invocation of it. Thus one would change the beginning of the config file to

```
WHIZZY_WIMP_CONFIGURED=0
```

and a check for WHIZZY_WIMP_CONFIGURED would be done before invoking the config file.

Modification of User Setup Files

In most situations, modifications to user setup files are both unneeded and dangerous. This tends to be the area where most sites do extensive customization, and it is the area where conflicts are most likely to occur⁵.

The already described method of a standard configuration file with a conditional invocation applies equally well to personal initialization files like .cshrc, .login, .profile, etc. Given the configuration method we have shown above, it should not be necessary. But if it is needed, do it right.

Modification of Existing System Files

We have already shown that it is not necessary to modify users personal setup files. Unfortunately, sometimes a package requires changes system boot files or configuration files. This can lead to some interesting problems. Curiously, the better-managed the site, the more likely the problem. Consider the following two real-world examples.

At the Industrial Technology Institute, we carefully placed all system configuration files under control of RCS. This has the same benefits for system management that it does for source code control. It also was a time bomb for one particular package which added two lines to

⁵ One CAD/CAM package which will remain unnamed requires over 300 lines of addition to .cshrc and .login files and includes multiple(!) lines like `set path = (a b c)`, utterly destroying the users carefully constructed path. Worse, the installing site consisted of ksh users. Worst, the lines to be added contained several bugs.

/etc/services and a 'paragraph' to /etc/rc.local. When the system manager went to modify one of those files, he checked out new copies from the RCS archive, made the changes, and installed the modified file, deleting the changes the install had made. Bringing back the install-time changes required restoring files from tape – not a favorite task of any administrator – and then adding them into the RCS archive. The problem occurred several times with several packages before it was diagnosed; now the managers routinely do `rcsdiff` even on what appear to be pristine files before checking out new copies for modification.

A similar problem occurred at sites which run file integrity checking packages for security or `rdist` [Cooper92] for distributed file management. In the first case, the integrity package begin reporting security violations on the `rc` files as soon as the package was installed. Recovery involved rebuilding the integrity database. In the second, the `rdist` update undid the install changes. Unlike the situation at ITI, the changed file was not backed up and hence not recoverable. A new install of the product was required.

The situation is further complicated when multiple packages are installed in quick succession. Package A modifies a file and renames it to `file.bak`. Package B modifies the same file and renames it to `file.bak`. The original file is now lost.

There is some simple methods for dealing with this. The install package should come with checksums for all system files for which it intends to modify. (Of course, this assumes the install has been checked in advance on all systems for which you are selling it.) A pristine system should be obtained and checksums computed for all files which will be modified. At install time, those files should be checksummed. If there is no match, it's extremely likely that either the file has changed or the install is being done on a new type of system. In either case, the install should halt and report the problem. *Blindly modifying already changed files is never acceptable.*

In addition, the scope of the changes should be strongly restricted. Adding ports or daemons to `inetd.conf`, `services`, and other simple tables is fairly simple and hard to do wrong. The various `rc` files are much more complex, and are discussed in a separate section below.

File permissions should be checked before making changes. RCS and other similar systems make the controlled file unwritable. Running as `root`, most packages simply ignore writability and blast away. *Check first!* If `inetd.conf` is permitted 444, the system manager is telling you something.

RC Files

An error in an `rc` file can make an system unbootable. In spite of this danger, a number of installation methods are quite cavalier about how they modify one or more of the `rc` files.

The failures largely fall into two groups: bugs in shell commands inserted into the `rc` file, and excessive changes to the file⁶.

System V `rc` files avoid the latter by isolating the changes into a directory of `rc` subfiles which are executed in an easily-specified order. The master `rc` file is careful about how it executes those files, and thus is resilient in the face of failure.

Berkeley-based systems are not so lucky. A number of changes have been proposed [Nieusma,Romig91,Simmons91], the reader should consult [Romig91] in particular for good suggestions. While the install package cannot make wholesale changes to the `rc` files, whatever changes are made should be done in accordance with the suggestions for good quality `rc` files.

In general, do not take the existing style or methods you see in the vendor-supplied `rc` as examples of how to do things. The quality of standard `rc` files is low.

⁶Although I have seen one install which simply mangled the `rc` file.

For the installer, the choice is a somewhat simpler one. Use the System V style of a number of small, individual `rc` files which are invoked by one of the master `rc` files. In the master, make a minimal change as shown in Figure 3.

Once again, we have defensive shell programming with checking for a file of the correct type. Some particular points to note:

- The use of mixed quotes in the first `echo` will reveal such programming errors as getting variable names wrong or putting spaces or newlines in the variable.
- Forcing all output (including errors) to `/dev/console` ensures that error messages get where they belong.
- Forcing the execution of the `rc` file into the background ensures that any errors in it will not affect the parent `rc` script.
- The check for the existence of the product-specific `rc` file ensures that the product and its `rc` file can be removed from the system without requiring modification of the master `rc` file.
- The entry both begins and ends with a comment such that its scope can easily be determined by the administrator. Good administrators do this with all their modifications to `rc` files; the install programs should do the same.

This last point, appropriate comments with begin and end markers, also applies to any other system tables which are modified. However, one should not assume those markers will still exist at uninstall time. One uninstall script looked for the begin marker, found it, and deleted everything from there to the end marker or end of file, whichever came first.

Diskless, Dataless and Diskful Nodes

Many installation scripts assume the hardware or software installed resides on the same machine as the install media or the execution of the install script. This is often not the case. It is not at all unusual for a systems `/usr` partition to be NFS-mounted read-only from one system, for `/usr/local` to be read-only from another, and for the installation media to be attached to a third system. These become formidable problems to which there is no general solution. The best one can do is design the package in such a way that installation can be performed in a series of steps

```
# Beginning of added material for WhizzyWimp
#
WW_RC_FILE="<>name of file here>"
if [ -x "${WW_RC_FILE}" ] ; then
    "${WW_RC_FILE}" > /dev/console 2>&1 &
else
    echo 'WhizzyWimp startup file "'${WW_RC_FILE}'" not found or' \
        > /dev/console 2>&1
    echo 'not executable.  Boot continues.' \
        > /dev/console 2>&1
fi
#
# End of added material for WhizzyWimp.
```

Figure 3

appropriate to the various environments.

Extracting data from remote media should be possible without requiring remote root access. An un-privileged account should be able to extract file sets using `tar`, `cpio`, or whatever is appropriate. The files containing the extracted sets can then be made read-accessible across the network to the install program.

There are four primary areas where files may need to be installed:

- The root partition. This is always writable on the local station by the local root.⁷
- The `/usr` partition. This is often mounted read-only from a master server, and requires root access on the master server.
- The `/usr/local` structure (which may not actually be local). This is sometimes local and sometimes read-only from another server.
- The install directory for most executables. This may be either local or remote.

Modifying each of these disk areas may require accessing the install media from four different systems, sometimes with and without root access. The only way to successfully deal with the problem is to break the install process internally into four steps which can be performed individually by the appropriate system managers if needed. Thus there would typically a separate set in the install media for each of the areas the install modifies.

Fortunately, installing device drivers onto diskless dataless workstations is an extreme case. On the other hand, it doesn't have to happen often to prevent sales of the particular hardware or software. The install software should be designed to detect which (if any) of these situations apply and generate the appropriate messages. A relatively small amount of effort results in an install which seems no different in the standard case but handles the extremes well.

Documentation Of Changes

A good installation will include complete documentation of all changes to the system. This documentation should cover files added, files changed, and preserved files.

Many packages include a `Manifest` file which lists all files installed. This is useful but should go much further. The manifest file should list all files, their ownerships and group memberships, permission modes, and a checksum of all files which should be invariant. Ideally there would also be a script included which the administrator could run at any time to determine if any files have changed and how, and note missing and added files. A number of existing installation methods (Digital Equipment, UNIX V, proposed POSIX standard) come quite close.

When system files are changed, three particular items should be documented. First, the reason for the change should be noted. This should be done in a `ReadMe.InstallNotes` file. The file should also include comments describing the manifest, the uninstall method, and the next two items.

Second, a copy of the original unmodified file should be kept. Great caution should be used here! It is not sufficient to copy `/etc/rc.local` to `<some-dir>/rc.local`, as subsequent reinstallation will wipe out the saved copy. The copied file should be named `rc.local.YYMMDD.HHMMSS` where `YYMMDD.HHMMSS` is the full date as printed by the standard UNIX command `date '+%Y%m%d.%T'`. This particular choice ensures that an `ls` of the directory will show all the files in age order. The truly paranoid will put the century as well.

Finally, a patch should be generated which will undo the change. As subsequent package installs and system modifications are made the patch will no longer work in its literal form.

⁷ Some configurations literally rebuild the root partition at every boot. In this case installation of software which requires modification of files in `/etc` is best left to the manual intervention of the site managers.

However, the patch file can easily be modified to change the line number and can become a tool to allow the experienced administrator to either undo the changes made or re-apply the changes if lost.

The typical inexperienced administrator or end user will never need or see the ReadMe files. Their purpose is not for day to day use, but for dealing with errors. The experienced administrator will look for it immediately if a problem is found; and the inexperienced will eventually stumble over it.

Dealing with multiple simultaneous installations

A properly designed software package should permit the simultaneous installation of multiple versions. This is often temporarily necessary for the conversion period between versions, and sometimes is needed indefinitely for the support of legacy systems. One of WhizzyWimps inspirations is a classic example of this. A read-only version of WhizzyWimp is available to be packaged with other software. This allows the package vendor to include very nice on-line documentation, but leads to interesting complications. The package vendor may support products on systems (eg, Sun 3s running SunOS 3.5) where up-to-date versions of WhizzyWimp will no longer run. This is fairly easy to deal with at the end user site, but a nightmare for the package vendor.

The use of the WhizzyWimp version settings in the configuration files is one way to handle the problem. The package vendor has multiple versions of WhizzyWimp installed, each neatly isolated into directories named after the version. Most users do not define a WhizzyWimp version number in their environment, and hence always get the latest version.

Another form of multiple simultaneous installation comes from having multiple processor and/or operating system types in a network. Here there are two methods, both of which work well. In both cases, one first isolates the executables for given OS/processor into appropriately named directories.

For the first case, can add a test to the configuration file to dynamically determine which OS and processor is in use and create the appropriate pathname. into particular directories. This complicates the configuration file somewhat, but is amenable to simple analysis and test.

The other is more subtle, and is taken from the FrameMaker installation. In this method, all scripts are symbolic links to a script called `.wrapper`. The `.wrapper` script dynamically determines OS and processor type, and invokes an executable from the appropriate directory. The name of the executable is taken from `$0` in the environment, almost always the same as the symbolic link to `.wrapper`.

As shipped by Frame, the `.wrapper` script is functional for FrameMaker but is not general enough to be used by most other packages. While it could be extended, there seems to be no significant benefit to using it over the simple configuration script method.

Uninstallability

For a product to be uninstalleable, there must be careful tracking of the location of all installed files and all modifications made to other system files. Here is where the manifest files, dated backup files and patch files come into their own.

The uninstall script should *not* blindly copy back the modified system files, nor should it ignore them. It should apply the patch file to a copy of the saved copy (thereby reversing the changes) and compare the result to the current installed copy. If they differ, there have been subsequent changes to the system file. These changes would be lost if the old versions of the system file were simply copied back. Instead, a message should be generated by the uninstall script stating

* The Frame `.wrapper` script cannot be reproduced here due to copyright restrictions.

that the files are restored to their pre-install version, and directing the uninstaller to the directory containing the patch and install ReadMe file.

Another caution for uninstallation is the presence of multiple versions. If one is simply removing a now-outdated version, it is not a good idea to delete the master shell script or the socket numbers from `/etc/services`.

Note, however, that it is perfectly safe to delete the executables and library files in the directory and the specialized `rc` file which is invoked by the master `rc` file. The modification done to the master `rc` file checks for the existence of the specialized `rc` file, prints an appropriate message, and continues with system boot.

Inevitably, someone will eventually notice the message and investigate. For ease in that investigation, the uninstall process should write a message in the directory with the ReadMe file informing future administrators of "pending" changes to the files.

System Security

The security issues which have already been discussed focused on the issues of modifying files in a distributed network. Unfortunately these are the easy issues.

Many packages require the use of a dedicated ID for file ownership and set user ID programs. This is a perfectly acceptable practice, but great caution should be used in the implementation.

One cannot rely on a given user login id or UID number to be available on any system. Name conflicts⁹ are inevitable and can happen on systems of any size. If your product is popular enough, you will eventually find one. A number of large sites are already facing exhaustion of UID space [Doster90] (as few as 32765 on some systems), so choosing a fixed UID number will inevitably involve a collision as well.

The only general solution is to allow the product login id to be modified by the site manager at install time and modified again later if needed. This modification would be made in the system install files. The running executable would then do the appropriate `getpwnam(3)` calls rather than depending on compiled-in parameters of any type.

With a user id defined for the package, one can do a great many useful things with `suid` programs which will not compromise system security. Unfortunately these require careful programming. A full discussion of this is outside of the scope of this paper. [Simmons90] examines many of these issues; a careful examination of the C News or INN source will yield useful examples. The `lpr` system is a classic bad example.

Many packages attempt to work around the issue by installing `suid` root programs. This should *never* be done on a wholesale basis. As system administrators become and sites in general become more concerned about security issues sites will simply refuse to install such programs as unacceptable risks.

Other packages take a different tack – they make all files globally writable, then depend on internal equivalents of access control lists to manage data integrity. Typically such packages have been ported from non-UNIX systems to UNIX and the developers were either ignorant of how UNIX file and `suid` systems worked or were not given sufficient time to do the correct work¹⁰.

⁹ Around 1982 the Los Angeles phone book contained a listing for the Ingres family, and the last name Root has caused interesting issues on some systems.

¹⁰ There is one popular mainframe accounting package which has been ported to UNIX which does exactly this. It has now been two years since the problem was reported and they have failed to fix it.

Adding IDs to a system is a surprising difficult task. At a minimum the installation script should ask before doing it; it is far preferable to have the administrator set up the id first.

As a final note on IDs, everything which has been said about user ids applies to group ids as well.

Some administrators will not blindly execute *any* script as root. Make your scripts straightforward, whenever possible using standard utilities.

Porting From Non-UNIX Systems

Many of the problems discussed can be laid at the feet of programs which have been ported from non-UNIX systems. These programs and their operation usually have many fundamental assumptions about how the operating system works. When the port effort starts, the developers and administrators take advantage of UNIX's customizability and make their UNIX systems look as much as possible like their original systems. This does minimize their own difficulties and seems to reduce the slope of the learning curve of a new OS. In reality, they have not gotten up the curve at all. They have merely delayed encountering it until the package is ready to install at a UNIX-savvy site. At that point the curve becomes a brick wall and the product falls apart.

The whole world is not UNIX, or VMS, or MVS. The techniques which have been discussed here are specific to UNIX, but their underlying principals are *general*. We are now ready to articulate them and how they relate to software development.

Localization of Product Information

A general interface for getting locations of files and the values of settable items should be developed. A generalized `getsetting()` function should be so that the high level interface is independent of the low-level implementation. Such an interface works equally well for UNIX environment variables, VMS logicals, data from flat files, or compiled-in tables. The underlying mechanism is unimportant. What is critical is an organized method of retrieving the data from a well-managed store rather than compiling values into multiple locations in a program.

Verification of Configuration

Once obtained, data should not be trusted until tested. Just as the configuration script checks to see if given directories exist, the initialization portion of the program should check the environment retrieved by `getsetting()` against the actual system environment. Far too many programs halt with the simple message "could not open library". This is inadequate. The error message should indicate which library, the name of the file, and the type of error (permission denied, missing file, etc).

Documentation

The documentation provided must be two-fold. First, standard documentation must be available for the administrator who is attempting to debug a possibly defective product installation. Without knowledge of what a proper installation is, the administrator can never be sure that the installation is correct.

Second, the installation process must provide some dynamic tracking of what it does and preserve that tracking in a reasonable location. Messages printed at install time are useful, but must be supplemented with logs, message files, and sometimes even recordings of the install process.

Conclusion

Two of the most important principles of good programming are information hiding and well-defined interfaces with disciplined use. These apply equally well to the installation of software. Follow them for the installation, and the resulting package will be improved.

Other Commentary

The previously referenced POSIX standard [Archer93] is a must. FTPable drafts are available from `cdm.jw.fnal.gov` in the directory `/posix/1387.2`. As of this writing, the 12th draft is now available.

An informal *Software Installation Workshop* was conducted by Paul Anderson at the 1992 Large Installation System Administration Conference. Notes from the workshop are in [Anderson93], and a mailing list formed as a result can be reached at `soft-managers-request@nas.nasa.gov`.

References

- [Anderson93]: Paul Anderson, *Software Installation On Large Systems*, **login:**, March/April 1993, Volume 18, No. 2.
- [Archer93]: Barrie Archer, *Towards a POSIX Standard for Software Administration*, Proceedings of the Large Installation Systems Administration Conference, 1993.
- [Collyer87]: Geoff Collyer and Henry Spencer, *News Need not be Slow*, Winter USENIX Conference, 1987.
- [Cooper92]: Michael A. Cooper, *Overhauling Rdist for the '90s*, Proceedings of the Large Installation Systems Administration Conference, 1992.
- [Cygnus93]: David MacKenzie, Roland McGrath, and Noah Friedman, **Autoconf: Generating Automatic Configuration Scripts**, Cygnus Support documentation.
- [Doster90]: William A. Doster, Yew-Hong Leong, and Steven J Mattson, *Uniqname Overview*, Proceedings of the Large Installation Systems Administration Conference, 1990.
- [Nieusma]: Posting of rewritten Sun `rc` files to `comp.sys.sun`.
- [Pixley92]: K. Richard Pixley, **On Configuring Development Tools**, Cygnus Support documentation.
- [Pixley93]: K. Richard Pixley, **Cygnus Configure**, Cygnus Support documentation.
- [Romig91]: Steve Romig, *Some Useful Changes for Boot RC Files*, Proceedings of the Large Installation Systems Administration Conference, 1991.
- [Salz92]: Rich Salz, *InterNetNews: Usenet transport for Internet sites*, Summer USENIX Conference, 1992.
- [Simmons90]: Steve Simmons, *Life Without Root*, Proceedings of the Large Installation Systems Administration Conference, 1990.
- [Simmons91]: Posting of rewritten Sun `rc` files to `comp.sys.sun` and Ultrix `rc` files to `comp.unix.ultrix`.
- [Wall]: Larry Wall, *metaconfig(1)* manual page, *dist2* package, posted in `comp.sources.unix`, Volume 16, 1988.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

- * sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
- * fostering innovation and communicating both research and technological developments,
- * providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its twice-a-year technical conferences, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *;login:*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with the MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *;login:*.

SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association, devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well. There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided.

USENIX Association membership services include:

- * Subscription to *;login:*, a bi-monthly newsletter;
- * Subscription to *Computing Systems*, a refereed technical quarterly;
- * Discounts on various UNIX and technical publications available for purchase;
- * Discounts on registration fees to twice-a-year technical conferences and tutorial programs and to the periodic single-topic symposia;
- * The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
- * The right to join Special Technical Groups such as SAGE.

Supporting Members of the USENIX Association:

ANDATACO	OTA Limited Partnership
ASANTÉ Technologies, Inc.	Quality Micro Systems, Inc.
Frame Technology Corporation	UUNET Technologies, Inc.
Network Computing Devices, Inc.	Enterprise System Management Corporations (SAGE supporting members)

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

Email: office@usenix.org
Phone: +1-510-528-8649
Fax: +1-510-548-5738

